Computer Shop

Ian Stewart
Robin Jones

ZX Spectrum Maschinencode



Springer Basel AG

Computer Shop
Band 8

Ian Stewart / Robin Jones

ZX Spectrum Maschinencode

Aus dem Englischen von Tony Westermayr

Springer Basel AG

Die Originalausgabe erschien 1983 unter dem Titel:
"SPECTRUM Machine Code"
bei Shiva Publishing Ltd., Nantwich, England

Professor Ian Stewart
Mathematics Institute
University of Warwick
Couventry CA4, 7AL
England, U.K.

Professor Robin Jones
Computer Unit
South Kent College of Technology
Ashford, Kent
England, U.K.

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Stewart, Ian:
ZX Spectrum Maschinencode /
Ian Stewart ; Robin Jones. Aus d. Engl. von
Tony Westermayr. – Basel ; Boston ; Stuttgart :
Birkhäuser, 1983.
(Computer-Shop ; Bd. 8)
Einheitssacht.: Spectrum machine code ⟨dt.⟩
ISBN 978-3-7643-1535-1
NE: Jones, Robin:; GT

Ursprünglich erschienen bei der deutschsprachigen Ausgabe: Birkhäuser Verlag, Basel 1983

Umschlaggestaltung: Bruckmann & Partner, Basel

ISBN 978-3-7643-1535-1 ISBN 978-3-0348-6735-1 (eBook)

DOI 10.1007/978-3-0348-6735-1

Inhalt

* siehe Beilagekarte

Vorwort

Das ist eine Einführung in den Z8∅-Maschinencode, eigens abgestellt auf den Sinclair Spectrum. Sie geht davon aus, daß Sie BASIC einigermaßen beherrschen, Maschinencode aber gar nicht, und führt Sie mit einfachen Beispielen und Projekten an den Punkt, wo Sie selbst Maschinencode-Routinen schreiben, sie innerhalb eines BASIC-Programms fahren, mit SAVE auf Kassette sichern und mit LOAD wieder zurückladen können. Eigentlich handelt es sich dabei um die Spectrum-Version der zweiten Hälfte unseres Buches "Maschinencode und besseres BASIC" (erschienen bei Birkhäuser, Basel), überarbeitet, um auf die besonderen Merkmale des Spektrum einzugehen, und mit Ergänzungen versehen, die mehr von den besonderen Fähigkeiten des Spectrum ins Spiel bringen. Wie wir in dem erwähnten Band schon feststellten, sind die Grundsätze für gutes Programmieren und Z8∅-Maschinencode bei *jedem* Computer, der einen Z8∅-Mikroprozessor verwendet, dieselben. Das stimmt auch, aber natürlich erleichtert es das Dasein, nach einer Beschreibung vorzugehen, die vollkommen auf die eigene Maschine abgestellt ist. Um also alles zu vereinfachen und Sie vor der Mühe zu bewahren, Listings anpassen zu müssen, ist das hier für Sie schon alles erledigt worden.

Der Hauptvorteil beim Maschinencode: Er kann eine Reihe von Aufgaben erfüllen, die mit BASIC zwar *möglich* sind, aber zu lange dauern. Der Hauptnachteil: Er verlangt viel mehr vom Programmierer, der gezwungen ist, den kleinen, exakten Details auf der Spur zu bleiben, wo genau in der Maschine die Information gespeichert wird, welche Form sie annimmt und wie der Spectrum sie auffaßt. Wenn Sie Maschinencode aber wirklich *lernen*, erfahren Sie auch viel Neues über ihren Computer!

Wir beginnen damit, daß wir eine "Theorie" aufstellen: Wie Zahlen im Computer gespeichert, wie negative Zahlen behandelt werden und wie binäre und hexadezimale Codes (die unverzichtbar sind) funktionieren. Als nächstes untersuchen wir die *vereinfachte* Version des Z8∅-Chips, um die Hauptprinzipien festzulegen (Register, Adressiermethoden, Indizieren und indirekte Steuerung, Programmzähler und Stapelzeiger) ohne uns Gedanken über pedantische Kleinigkeiten machen zu müssen. Wenn Sie gleich mit dem Z8∅ anfangen, muß jede Anweisung durch Wenn und Aber und Vielleicht ergänzt werden. Der Mikroprozessor ist ein kompliziertes Gebilde, und es fällt viel leichter, zu verfolgen, wie es funktioniert, wenn man Vergleiche mit einfacheren Dingen ziehen kann.

Anschließend wird der Z8∅ selbst beschrieben und eine wichtige Gruppe von Befehlen – die LOAD-Befehle – im Einzelnen besprochen. Diese Gruppe wird dazu benützt, die verschiedenen "Adressiermethoden" in Maschinencode zu erklären.

Wir erläutern, wie man Maschinencode speichert, fährt, sichert und lädt, und entwickeln ein BASIC-Programm, um alle diese Aufgaben so leicht wie möglich zu machen. Alle späteren Programme in diesem Band stützen sich auf dieses BASIC-Programm.

Zu den besprochenen Routinen gehört ein Maschinencode-Mulitplikator, der viele wichtige Befehle beispielhaft darstellt. Wir beschreiben im Einzelnen, wie das Spectrum-Display gesteuert wird und wie es verändert werden

kann; eigene Kapitel untersuchen nützliche Maschinencode-Routinen für Attribut- und Displaydatei (einschließlich Vorgänge wie Abrollen, Farben verändern, Mustererzeugung, FLASH ein- und ausschalten).

Die nützlichsten Flaggen werden anhand einer Zeilen-Umnumerierungsroutine genauer dargestellt. Zwei leistungsstarke Befehlsgruppen, Blocksuche und Blockübertragung, werden vorgestellt; letztere bezieht sich auf verschiedene Abrollroutinen. Ein abschließendes Kapitel befaßt sich mit ein paar beachtenswerten Ergänzungen.

Der Anhang enthält mehrere Tabellen nützlicher Information für das Programmieren in Maschinencode: Umrechnung Hex in Dezimal, Speicherreservierung, Systemvariable in Hex, die Z8Ø-Befehle, ihre Auswirkungen auf Übertragungs- und Nullflaggen, Hexcodes (damit es für Sie praktischer ist, alphabetisch aufgeführt), und einen nützlichen Teilassembler (HELPA), zur leichten Abänderung in BASIC geschrieben, mit dem Sie Maschinencode auf einigermaßen qualfreie Weise schreiben, redigieren und fahren können. Vor allem berechnet er relative Sprünge automatisch, was eine Menge umständlicher Berechnungen erspart (die gern danebengehen, wenn man sie mit der Hand macht, und ein Chaos hervorrufen).

Das Buch ist so geschrieben, daß Sie, wenn Sie wollen, die Maschinencode-Routinen gut nützen können, *ohne* zu verstehen, wie sie funktionieren. Aber wir hoffen, daß Sie mehr erstreben, nämlich, zu lernen, wie man *selbst* Maschinencode verfaßt.

Bis jetzt haben wir uns als "wir" bezeichnet, stellten aber wie in unseren anderen Büchern fest, daß das später nicht mehr so gut funktioniert. Von jetzt an sprechen wir von uns also per "ich". Wenn wir wirklich einmal "wir" sagen, heißt das "der Leser und ich". Das mag ein bißchen albern klingen, aber in Wahrheit ist es so informativer.

Für Hinweise auf Druckfehler ist der Verlag jederzeit dankbar. Bitte wenden Sie sich an: Birkhäuser Verlag AG, Lektorat, Postfach, CH-4010 Basel (Schweiz).

1 Appetitanreger

Nur um zu beweisen, daß Maschinencode wirklich Dinge leistet, die in BASIC nicht möglich sind, hier ein Programm, mit dem man spektakuläre, wenn auch recht sinnlose, Displays erzeugen kann.

Sie hätten dieses Buch nicht gekauft oder würden es in ihrem Buchladen nicht durchblättern, wenn sie nicht gehört hätten, daß der Spectrum bemerkenswerte Dinge sehr schnell bewältigen kann mit etwas, das sich *Maschinencode* nennt. Das stimmt zwar, aber der Haken beim Maschinencode ist der, daß er Ihnen im Gegensatz zu BASIC das Denken nicht abnimmt. Sie müssen viel mehr auf die kleinen Details achten und im Auge behalten, wo genau in der Maschine Ihr Code sitzt. Maschinencode ist ganz entschieden nicht "anwenderfreundlich", sieht am Anfang eher nach ägyptischen Hieroglyphen aus und hat den Reiz und die sofortige Verständlichkeit eines Telefonbuchs in Urdu-Sprache.

Ganz so schlimm ist es aber auch wieder nicht, und Sie werden das bald gepackt haben. Auf jeden Fall müssen Sie sich ganz gehörig anstrengen, bevor es wirklich lohnt. Um Sie davon zu überzeugen, daß das der Mühe wirklich wert ist, will ich deshalb mit ein paar Maschinencode-Routinen anfangen, die auffällige, sehr schnell bewegliche Displays abstrakter Art erzeugen. Wenn Sie das mit BASIC schaffen, gehören Sie offenkundig zu denen, die alle Probleme der Weltwirtschaft im Kopf noch vor dem Frühstück lösen (das zweifellos aus einer dreifachen Portion Haferflocken besteht . . .).

Geben Sie sich noch keine Mühe, alles zu verstehen; das kommt später. Tippen Sie einfach ab und fahren Sie mit RUN. Sie brauchen ein paar Tasten, die Sie wahrscheinlich noch nicht oft verwendet haben, vor allem

USR (Taste "L" im erweiterten Modus)
CLEAR (Taste "X" im Schlüsselwort-Modus)

und

POKE (Taste "O" im Schlüsselwort-Modus)

Hier das erste Programm:

```
1Ø   CLEAR 31999
2Ø   BORDER Ø
3Ø   DATA 1, Ø, 3, 17, Ø, 88, 33, Ø, Ø, 237, 176, 2Ø1
4Ø   FOR i = Ø TO 11
5Ø   READ x
6Ø   POKE 32ØØØ + i, x
7Ø   NEXT i
8Ø   PAUSE Ø
9Ø   LET y = USR 32ØØØ
```

Achten Sie darauf, das richtig einzugeben, vor allem die DATA-Liste. Fahren Sie nun mit RUN. Der Schirm bleibt leer, bis Sie eine Taste drücken (wegen Zeile 8Ø). Sobald Sie eine drücken, wird *sofort* reagiert, und der Bildschirm füllt sich mit bunten Kästchen, von denen manche blinken. (Wenn nicht, gehen Sie das Listing noch einmal durch. Sie müssen vielleicht erst den Stromstecker ziehen, um zurückzustellen – das ist einer der Haken beim Maschinencode.)

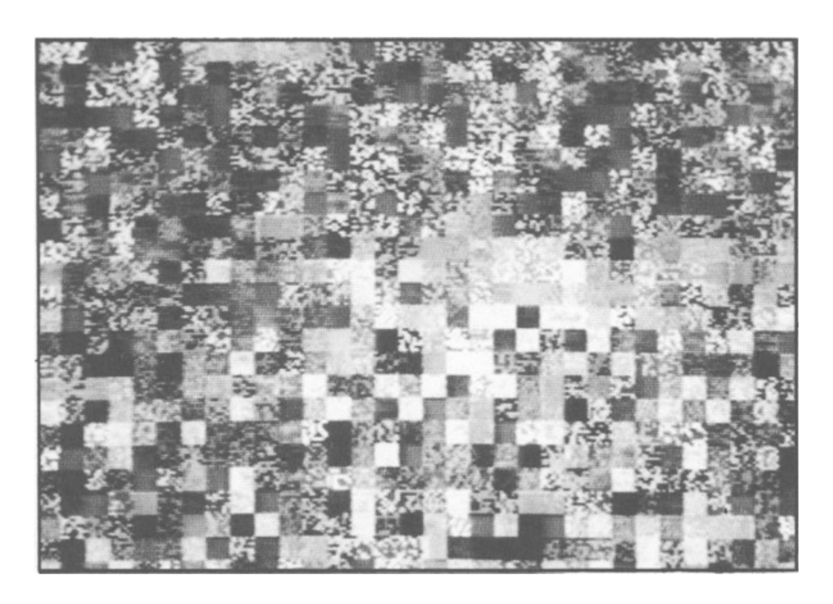


Abbildung 1.1
Seitlich abrollende Kästchen in Zufallsfarben, manche gefleckt, manche hell, manche blinkend.

Das läuft schnell, ist aber noch nicht sehr dramatisch. Der nächste Schritt besteht darin, im Programm ein paar Veränderungen vorzunehmen. Löschen Sie Zeile 9Ø und fügen Sie an:

```
1ØØ  LET t = Ø: LET s = Ø
11Ø  IF t> = 256 THEN LET t = t – 256 * INT (t/256): LET s = s + 1
12Ø  POKE 32ØØ7, t: POKE 32ØØ8, s
13Ø  LET y = USR 32ØØØ
14Ø  LET t = t + 1
15Ø  GO TO 11Ø
```

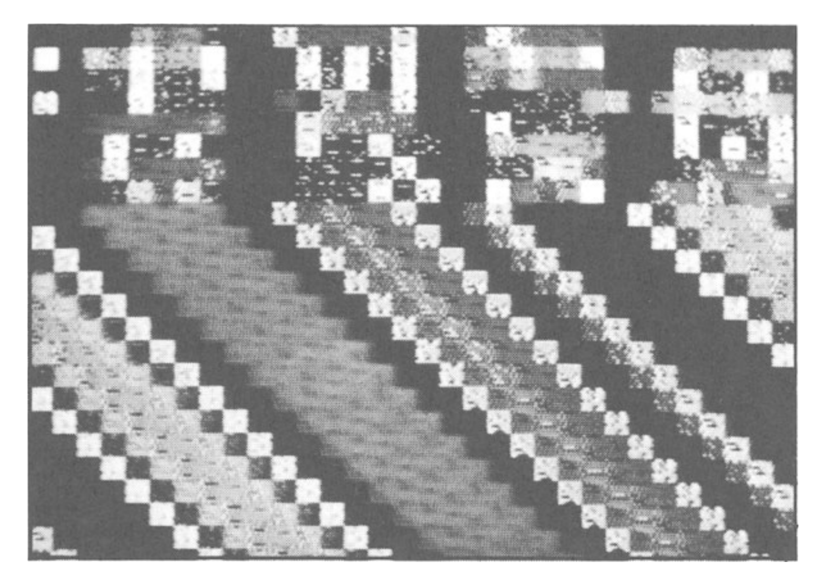


Abbildung 1.2
... zeigt jetzt mehr Struktur – schnell rotierende Streifenmuster ...

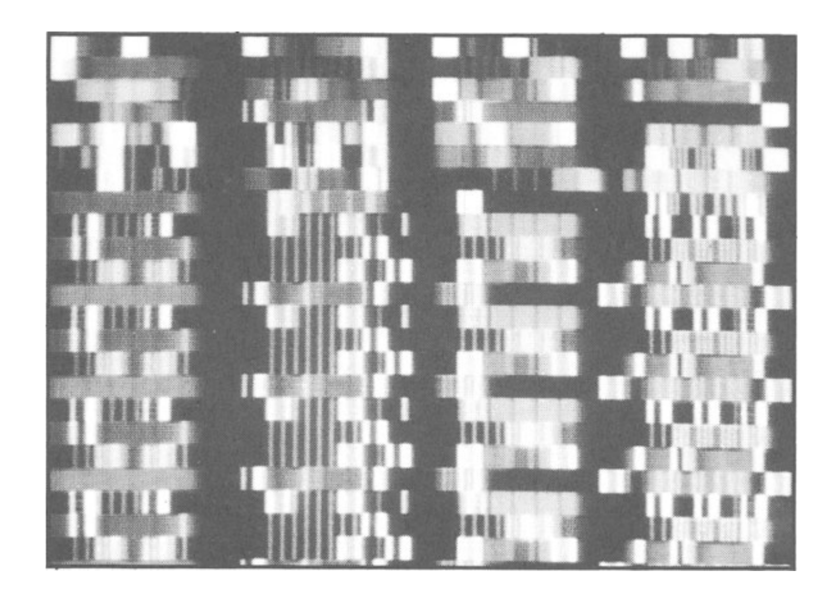


Abbildung 1.3
... wetten, daß Sie das nicht bemerken! Die sausen so schnell vorbei ...

Drücken Sie RUN und auf irgendeine Taste, damit es losgeht. Sie erhalten ein ähnliches Display, das jetzt aber mit einer vernünftigen Abrollgeschwindigkeit *seitwärts* läuft. Verändern Sie Zeile 14∅ zu:

```
14∅   LET t = t + 32
```

dann rollt es aufwärts; verändern Sie zu

```
14∅   LET t = t + 31
```

und es rollt *diagonal* ab. Probieren Sie, t ein paar andere Werte zu geben.

Machen Sie aus 14∅ nun wieder LET t =t + 1, verändern Sie die DATA-Anweisung zu

```
3∅   DATA 1, ∅, 24, 17, ∅, 64, 33, ∅, ∅, 237, 176, 2∅1
```

und wiederholen Sie das Ganze. Ich will Ihnen nicht sagen, was Sie erwartet – sehen Sie selbst! Ändern Sie Zeile 140 ab wie vorher.

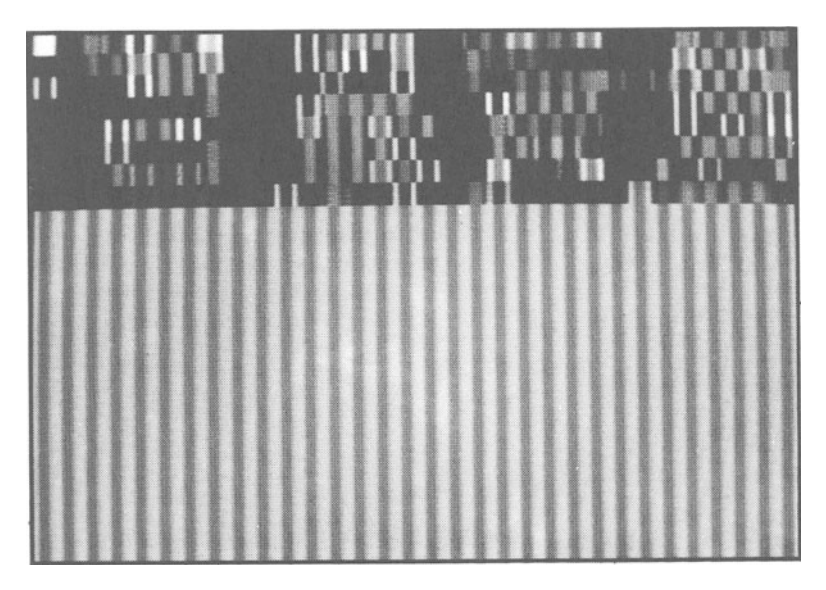


Abbildung 1.4
. . . jetzt eine Pause mit zarten grünen Streifen . . .

Licht-Schau

Ungewöhnlich, mag sein, möglicherweise noch nicht spektakulär. Nun fügen wir beide Routinen zusammen und stellen noch ein bißchen feiner ein:

```
 1Ø  CLEAR 31999
 2Ø  BORDER Ø
 3Ø  DATA 1, Ø, 24, 17, Ø, 64, 33, Ø, Ø, 237, 176, 2Ø1
 35  DATA 1, Ø, 3, 17, Ø, 88, 33, Ø, Ø, 237, 176, 2Ø1
 4Ø  FOR i = Ø TO 23
 5Ø  READ x
 6Ø  POKE 32ØØØ + i, x
 7Ø  NEXT i
1ØØ  LET t = Ø: LET s = 6Ø
11Ø  IF t > = 256 THEN LET t = t – 256 * INT (t/256): LET s = s + 1
12Ø  POKE 32ØØ7, t: POKE 32ØØ8, s
125  POKE 32Ø19, t: POKE 32Ø2Ø, s – 1
13Ø  LET y = USR 32ØØØ
14Ø  LET y = USR 32Ø12
15Ø  LET t = t + 1
16Ø  GO TO 11Ø
```

Fahren Sie das mit RUN und lassen Sie es ein, zwei Minuten laufen. Es geht ganz friedlich an, aber dann ist auf einmal der Teufel los . . .

Verändern Sie Zeile 15Ø wie vorher zu t + 32, t + 31 etc.

Ändern Sie die Initialisierungen in Zeile 1ØØ. Was geschieht, wenn Sie mit s = Ø oder s = 4Ø beginnen?

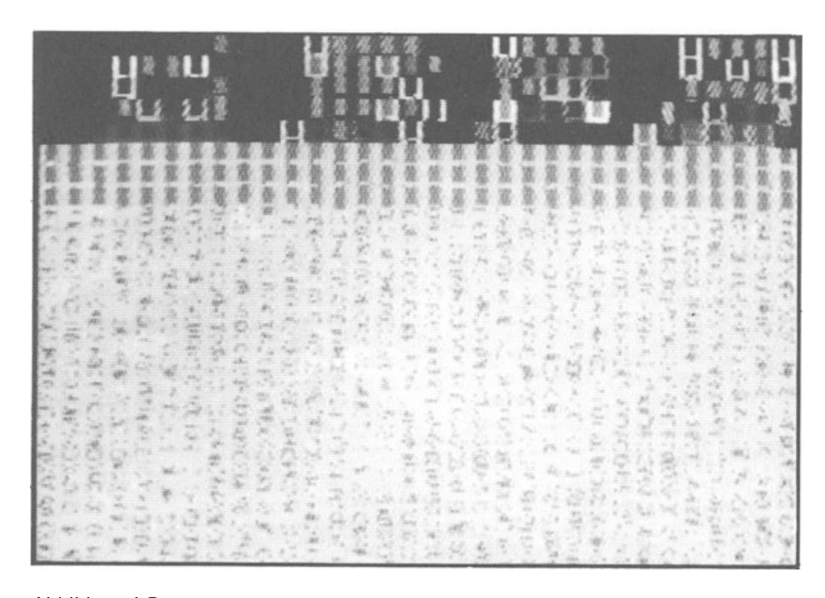


Abbildung 1.5
. . . da kommen sie wie Schwärme wütender Insekten und fressen alles auf, was sich in den Weg stellt!

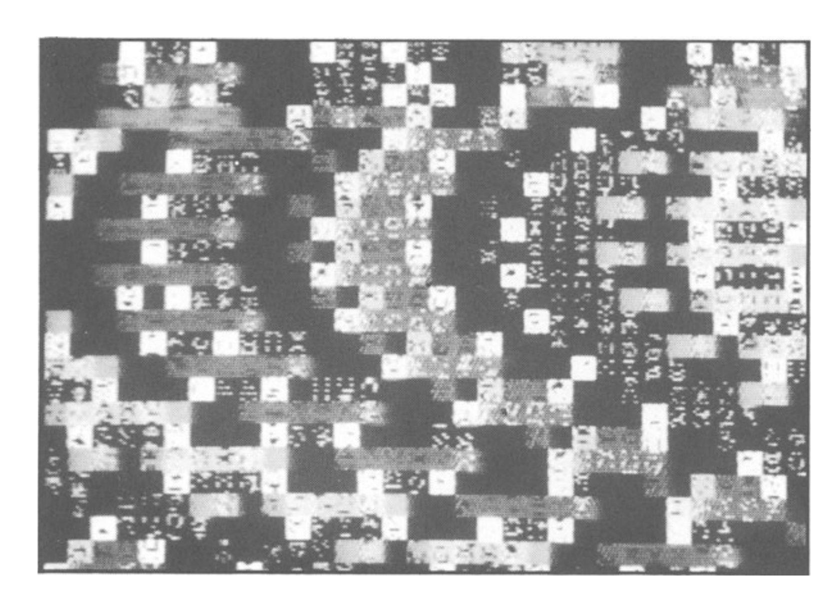


Abbildung 1.6
. . . und weiter in dichte Farbwirbel. Alles durch zwei Dutzend Bytes Maschinencode (und 16K ROM). Dabei sind das nur Beispiele!

Um s = 63 geht es so schnell zu, daß man es kaum noch verfolgen kann. Wenn Sie die Zeile anfügen

```
145 IF INKEY$ = "" THEN PAUSE Ø
```

werden Sie feststellen, daß gar nichts passiert, wenn Sie keine Taste drücken. Sie können das Programm jederzeit in "Einzelstufen" laufen lassen, wenn Sie die Taste rasch drücken, den Finger wegnehmen und wieder drücken. Sie werden viele Muster sehen, die bei dem vorherigen Schnelldurchgang zu rasch vorbeigesaust sind.

Sie können für dieses Programm endlose Variationen erfinden. Nur die Zeilen 3Ø, 35, 13Ø oder 14Ø, wo der Maschinencode behandelt wird, dürfen Sie nicht anrühren.

Ich hoffe, Sie sind jetzt davon überzeugt, daß Maschinencode mehr zu bieten hat. Allerdings: Sieht man von der Erzeugung hübscher Muster mit hoher Geschwindigkeit ab, würde die Behauptung schwerfallen, daß speziell dieses Beispiel etwas besonders Nützliches leistet. Sein Vorteil besteht darin, daß es ein sehr kurzes Stück Maschinencode mit großer Wirkung ist. Um Maschinencode-Routinen wirklich nutzen zu können, müssen wir sie auf eine richtig strukturierte Weise schreiben können und auf ein bestimmtes Ziel abstellen (genau wie bei einem guten BASIC-Programm). Der Rest dieses Buches ist diesem Ziel gewidmet.

2 Zahlen in Maschinencode

Das binäre System ist ein bißchen so, als zählten Sie statt mit den Fingern mit den Füßen. Für Maschinencode brauchen Sie nicht mehr als sechzehn Füße.

Bei Zahlen denken wir normalerweise in Zehnerbegriffen. Wenn ich die Zahl 3814 hinschreibe, verstehen wir das alle als

3 × 1ØØØ + 8 × 1ØØ + 1 x 1Ø + 4 x 1

und wir können sehen, daß wir, um einen "Stellenwert" von dem rechts daneben zu erhalten, ihn einfach mit Zehn multiplizieren. Wir sagen, die Zahl hätte die *Basis* Zehn.

Da wir das schon tun, seitdem wir denken können, fällt die Einsicht schwer, daß man dasselbe auch auf andere, völlig vernünftige Weise bewältigen kann. Die ersten Computerkonstrukteure kamen jedenfalls nicht darauf; sie verwendeten bei ihren Maschinen Zehnerdarstellungen und stießen auf ein paar üble Schwierigkeiten. Meistens rührten sie davon her, daß elektrische Verstärker sich nicht bei allen Signalen, die man eingeben will, gleich verhalten. Beispiel: Ein Verstärker, der sein Eingabesignal als Ausgabe verdoppeln soll, mag das bei Eingaben von 1, 2, 3 und 4 Einheiten durchaus tun, aber dann "flacht er ab", so daß eine Eingabe von 5 eine Ausgabe von nur 9.6 erbringt, 6 zu 1Ø.8 führt und man den Unterschied der Ausgaben für die Eingaben 8 und 9 kaum noch unterscheiden kann.

Legen Sie eine Musikkassette in Ihren preiswerten Kassettenrecorder und drehen Sie die Lautstärke auf. Hören Sie die Verzerrung bei den lauten Stellen? Das ist derselbe Effekt.

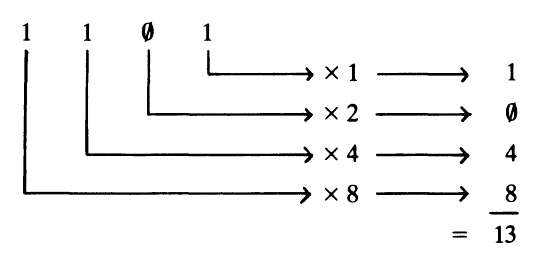
Die ersten Computerkonstrukteure hörten keine Verzerrung; sie stellten nur fest, daß die Maschine manchmal nicht zwischen verschiedenen Ziffern unterscheiden konnte, für einen Computer eine aussichtslose Sache. Sie mußten ihre Zahlendarstellung also neu überdenken, um sich dem anzupassen, was die elektronischen Geister am besten konnten.

Das Einfachste, was man mit einem elektrischen Signal machen kann, ist es an- oder abzuschalten; so kann man die Ziffern Ø (aus) und 1 (ein) zufriedenstellend darstellen. Verzerrung spielt keine Rolle mehr. Ob ein Signal vorhanden ist oder nicht, steht klar fest, ohne Rücksicht darauf, wie verunstaltet es sein mag. Aber können wir ein Zahlensystem erfinden, das nur Ø oder 1 verwendet?

Ja. Bei einer Zahl mit Basis Zehn ist die größtmögliche Ziffer 9. Addieren Sie 1 zu 9, und Sie haben 1Ø – stattgefunden hat ein *Übertrag.* Wir können jede Zahl mit jeder anderen Basis schreiben, die wir uns aussuchen, und die größtmögliche Ziffer wird stets eins weniger sein als die Basis. Bei Basis 2 ist die größte Ziffer 1, also enthält eine Zahl der Basis 2 (oder *binäre* Zahl) nur Ø und 1.

Wie ist es mit den Stellenwerten? Im Fall der Basis Zehn haben wir sie erhalten, indem wir bei 1 (rechts) anfingen und jedesmal, wenn wir um eine Stelle nach links gingen, mit 1Ø multiplizierten. Bei einer Binärzahl fangen wir auch bei 1 an, multiplizieren aber mit jeder Bewegung nach links mit 2.

So kann etwa die Binärzahl 11∅1 auf folgende Weise in Basis 1∅ verwandelt werden:



In die andere Richtung umzuwandeln ist genauso leicht. Nehmen wir als Beispiel 25. Wenn wir die binären Stellenwerte niederschreiben:

32 16 8 4 2 2 1

und von der linken Seite ausgehen, ist klar, daß wir eine 16 brauchen, so daß 9 übrigbleibt, die aus 8 und 1 besteht. Demnach ist 25:

∅ 1 1 ∅ ∅ 1

Hexadezimalcode

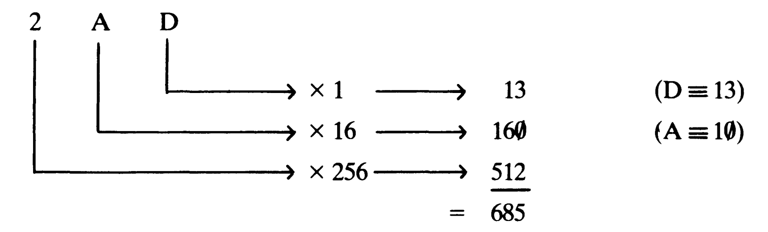
Das ist sehr schön bei relativ kleinen Werten, wird bei großen aber eher umständlich. Es gibt eine Reihe von schnellen Umwandlungsmethoden, und in "Sinclair ZX Spectrum – Programmieren leicht gemacht" von Birkhäuser stehen Programmlistings für die Umrechnung Binär/Dezimal und Dezimal/Binär, aber hier möchte ich eine Prozedur untersuchen, die den *Hexadezimal*code nutzt, weil uns das später gute Dienste leisten wird.

Eine Zahl in Hex (kein Mensch sagt jemals "hexadezimal", außer ich eben) ist eine Zahl mit der Basis 16. Stellenwerte werden also durch aufeinanderfolgende Multiplikationen mit 16 erreicht. Die ersten fünf sind:

65536 4∅96 256 16 1

"Moment mal!" höre ich von allen Seiten. "Das sind schlimme Zahlen, und außerdem ist bei Basis 16 die höchste Ziffer 15. Langsam wird das kompliziert."

Nur Geduld. Das Problem von Ziffern über 9 bewältigen wir dadurch, daß wir den Werten 1∅–15 die Buchstaben A–F zuteilen. Die Zahl 2AD in Hex läßt sich demnach auf folgende Weise in eine Dezimalzahl verwandeln:



Nun zum Angenehmen bei Hex. Da 16 einer der binären Stellenwerte ist (nämlich der fünfte) ergibt sich, daß jede Hexziffer einer Zahl durch die vier Binärziffern ersetzt werden kann, die sie darstellen. (Übrigens: Es dauert genauso lang "binary digit" – zu deutsch Binärziffern – zu sagen, wie "hexadezimal", weshalb der Ausdruck zu *bit* abgekürzt wird.) Die nächste Tabelle zeigt die Umwandlungen.

Dezimal	Hex	Binär
Ø	Ø	ØØØØ
1	1	ØØØ1
2	2	ØØ1Ø
3	3	ØØ11
4	4	Ø1ØØ
5	5	Ø1Ø1
6	6	Ø11Ø
7	7	Ø111
8	8	1ØØØ
9	9	1ØØ1
1Ø	A	1Ø1Ø
11	B	1Ø11
12	C	11ØØ
13	D	11Ø1
14	E	111Ø
15	F	1111

Eine ausführlichere Tabelle finden Sie in Anhang 1. (Zur Beachtung: In diesem Buch, das für den Spectrum geschrieben ist, können die Buchstaben groß oder klein geschrieben sein, Eingaben unterstellen aber, weil das praktischer ist, *Klein*buchstaben.)

Nehmen wir nun an, wir wollen 9Ø41 in Hex umwandeln. Zuerst ziehen wir zweimal 4Ø96 heraus, dann ein paarmal 256 und so weiter, nämlich so:

```
               9041
2 × 4096 =     8192 –
               ----
                849
3 ×  256 =      768 –
               ----
                 81
5 ×   16 =       80 –
               ----
                  1
1 ×    1 =        1 –
               ----
                  0
```

Die Hexzahl ist also 2351.

Nun schreiben wir die Ziffercodes einfach aus der Tabelle ab:

2	3	5	1
0010	0011	0101	0001

und das ist die binäre Entsprechung zu 9041; Sie setzen die vier Blöcke nur noch zusammen und erhalten 0010001101010001.

Die Umrechnung hex/binär ist so einfach, daß wir sehr oft Zahlen in Hex selbst dann belassen, wenn wir sie letztlich in binär brauchen. Schließlich fällt es nur allzu leicht, beim Schreiben langer Folgen von 0 und 1 Fehler zu machen.

Umwandlung durch den Computer

Hier folgt ein Programm, das Dezimal- in Hexzahlen umrechnet. Es teilt die Zahl fortlaufend durch 16 und sieht sich den Rest jedesmal an, zieht Ziffern also in der umgekehrten Reihenfolge zur oben gezeigten heraus.

```
 20  LET p = 4
 30  LET h$ = "0000"
 40  INPUT "Dez. Nr. (max 65535)"; tn
 50  LET n = INT (tn/16)
 60  LET r = tn – 16 * n
 70  LET h$ (p) = CHR$ (r + 48 + 39* (r> 9))
 80  LET tn = n
 90  LET p = p – 1
100  IF tn > 0 THEN GO TO 50
110  PRINT "Hexwert ist:"; h$
```

Zeile 7Ø sorgt dafür, daß die Ziffern und die Buchstaben a–f (wir verwenden lieber *Klein*buchstaben) ausgewählt werden. Das sieht ein bißchen verwirrend aus, weil der ASCII-Code, den der Spectrum zur internen Zeichendarstellung verwendet, sich auf eine ungünstige Weise verhält. Wir wollen zählen ... 7, 8, 9, a, b, ... und es wäre nett, wenn der Code für "a" größer wäre als für "9". Leider ist er um 4Ø größer, also um 39 zu hoch. Aus diesem Grund müssen wir 39 für Zeichen anfügen, die größer sind als 9. Der logische Ausdruck "r > 9" hat, wenn wahr, den Wert 1, und wenn falsch, Ø; die zusäzlichen 39 werden also dazuaddiert, wenn das Zeichen im Bereich a–f liegt. (Die 48 sind notwendig, weil für Ø der ASCII-Code 48 lautet.)

Das Resultat wird stets dargestellt als eine vierstellige Zahl mit vorangestellten Nullen, wo das erforderlich ist, die Buchstaben kleingeschrieben. Dies deshalb, weil wir Hexstellen stets kleingeschrieben *eingeben*, so daß dies als Merkstütze dient. Das Programm funktioniert nicht, wenn das Resultat mehr als vier Hexstellen enthalten sollte, aber das ist für unsere Zwecke wegen der Grenzen für die Speichergröße im Spectrum nur gut – erforderlich sind ohnehin nur vier Hexstellen.

Hier der Code für die umgekehrte Umrechnung (Hex in Dezimal):

```
14Ø  INPUT "4stellige Hexzahl eingeben"; h$
15Ø  LET tn = Ø
16Ø  FOR p = 1 TO 4
17Ø  LET tn = tn * 16 + CODE h$ (p) – 48 – 39 * (h$ (p) > "9")
18Ø  NEXT p
19Ø  PRINT "Dezimalwert ist:"; tn
```

Wieder gibt es einen Trick, die richtigen Werte in Zeile 17Ø zu generieren, die lediglich eine Umkehrung von Zeile 7Ø ist.

Wir könnten diese Routinen mit einem kleinen Menü zusammenfügen:

```
 2  PRINT "Umwandlung Dez/Hex"
 3  PRINT "1) DEZ– > HEX
          2) HEX– > DEZ
          3) ENDE"
 5  INPUT "1, 2 oder 3 eingeben"; ausw
 8  IF ausw = 1 THEN GO SUB 2Ø
 9  IF ausw = 2 THEN GO SUB 14Ø
1Ø  IF ausw = 3 THEN STOP
12  PAUSE Ø: GO TO 2
```

Selbstverständlich brauchen wir bei den Zeilen 12Ø und 2ØØ jeweils RETURN.

3 Positiv und negativ

Für den Umgang mit negativen Zahlen hat die Maschine einen raffinierten Trick parat.

Nachdem wir jetzt etwas über den Umgang mit Binärzahlen gesehen haben, wollen wir uns wieder damit befassen, wie sie innerhalb der Maschine behandelt werden. In der Regel wird eine Zahl mit einer festen Zahl von Bits festgehalten, oft 16 oder 24 oder 32, je nach Auslegung des Computers. Diese Bitzahl nennt man die *Wortgröße* des Geräts.

Sehen wir uns an, welche Zahlen in einem Wort von 4 Bits festgehalten werden können:

4 Bit-Muster	Dezimalwert
ØØØØ	Ø
ØØØ1	1
ØØ1Ø	2
ØØ11	3
Ø1ØØ	4
Ø1Ø1	5
Ø11Ø	6
Ø111	7
1ØØØ	8
1ØØ1	9
1Ø1Ø	1Ø
1Ø11	11
11ØØ	12
11Ø1	13
111Ø	14
1111	15

Warum in der Praxis größere Wörter genommen werden, ist offenkundig; eine Maschine, die nur die Zahlen Ø bis 15 darstellen kann, ist kaum ausreichend. Es gibt aber noch zwei andere Probleme. Die Schreibweise kann keine Bruchwerte (etwa 7.14) und keine negativen Zahlen darstellen.

Mit dem Problem der Brüche geben wir uns nicht ab, weil die meisten Maschinencode-Routinen nur ganze Zahlen verwenden, aber die Art, wie negative Zahlen behandelt werden, ist wichtiger.

Die Methode ist einfach: Wenn Sie die Binärdarstellung einer positiven Zahl haben und die negative Entsprechung hervorbringen wollen, tun Sie zwei Dinge:

1 Wandeln Sie jede Ø in eine 1 und jede 1 in eine Ø um. Man spricht hier vom "Kippen der Bits".
2 Addieren Sie zu dem Ergebnis 1.

Nehmen wir an, Sie wollen −3 haben.

3 = ØØ11 (4 Bit-Wort)
Wenn Sie die Bits kippen, haben Sie: 11ØØ
Dann 1 dazu: + 1
11Ø1

11Ø1 steht also für −3. Man spricht vom *Zweierkomplement* zu ØØ11.

Ich will nicht im Einzelnen erklären, wie das funktioniert, aber Sie können sich selbst beweisen, daß es in jedem Fall wie dem folgenden klappt:

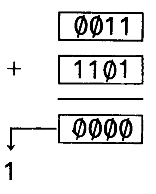
Wenn wir 3 zu −3 addieren (oder 5 zu −5 oder irgendeine Zahl zu ihrem Minuswert) müßten wir Null erhalten. Demnach:

ØØ11 (= 3)
\+ 11Ø1 (= −3)
= 1ØØØØ
111 Überträge

(Vergessen Sie nicht, daß in Binär 1 + 1 = Ø Übertrag 1!)

Wir erhalten also *keineswegs* ØØØØ, aber die niederwertigen 4 Bits *sind* Null, und wenn wir ein 4 Bit-Wort einfügen, fällt das höherwertige Bit am Ende einfach herunter. (Wenn Sie einen geeigneten Vergleich suchen: Denken Sie an einen Kilometerzähler im Auto mit drei Stellen; wenn er 999 erreicht hat und Sie einen Kilometer mehr fahren, zeigt er ØØØ an. An der linken Seite ist eine "1" "heruntergefallen").

Mit anderen Worten, wir hätten das so sehen sollen:



Das funktioniert immer, vorausgesetzt, die Zahl der Bits steht durchgehend fest. Vergessen Sie nicht, Nullen voranzusetzen, um die Zahl der Bits auf diese Normlänge zu bringen, *bevor* Sie das Zweierkomplement nehmen.

Schreiben wir die 4 Bit-Werttabelle um, in dem wir negative Zahlen aufnehmen:

Dezimal	Binär	Zweierkomplement	Dezimal
Ø	ØØØØ	ØØØØ	Ø
1	ØØØ1	1111	– 1
2	ØØ1Ø	111Ø	– 2
3	ØØ11	11Ø1	– 3
4	Ø1ØØ	11ØØ	– 4
5	Ø1Ø1	1Ø11	– 5
6	Ø11Ø	1Ø1Ø	– 6
7	Ø111	1ØØ1	– 7
8	1ØØØ	1ØØØ	– 8
9	1ØØ1	Ø111	– 9
1Ø	1Ø1Ø	Ø11Ø	–1Ø
11	1Ø11	Ø1Ø1	–11
12	11ØØ	Ø1ØØ	–12
13	11Ø1	ØØ11	–13
14	111Ø	ØØ1Ø	–14
15	1111	ØØØ1	–15

Sofort erkennen wir ein Problem: Jedes Bitmuster tritt zweimal auf, so daß beispielsweise 1ØØ1 9 oder auch –7 bedeuten kann. Wir müssen den Wertbereich also noch weiter einschränken. Ich habe um den Bereich, den wir tatsächlich darstellen wollen, eine gestrichelte Linie gezogen. Wenn Sie sich das höherwertige (ganz linke) Bit in jedem Muster ansehen, wird Ihnen auffallen, daß es bei einer positiven Zahl "Ø" und bei einer negativen "1" ist. Offenkundig eine sehr praktische Unterscheidung.

Der Zahlenbereich, den wir in einem 4 Bit-Wort unterbringen können, reicht also von –8 bis +7. Bei 5 Bits wären es –16 bis +15. Bei 6 Bits sind es –32 bis +31, und so weiter.

Ein 16 Bit-Wort (von Bedeutung, was den Z8Ø angeht) umfaßt den Bereich –32768 bis +32767. In Anhang 1 (Seite 1Ø9) finden Sie eine Tabelle der Zweierkomplemente für 8 Bit-Wörter.

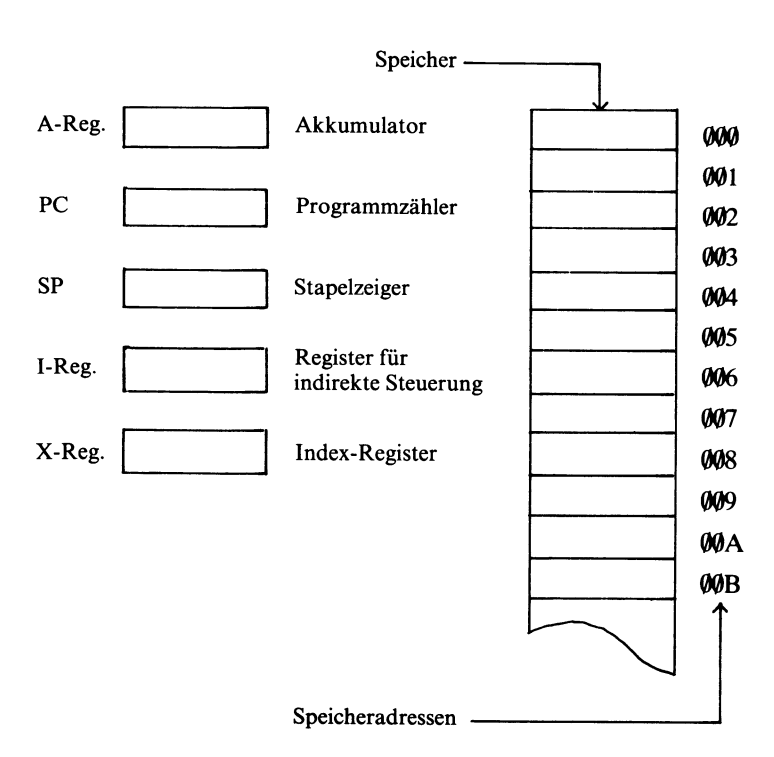
4 Maschinenarchitektur

Es fällt leichter, mit einer vereinfachten, imaginären Maschine anzufangen. Der Z8Ø-Mikroprozessor hat Ähnlichkeit damit, ist aber komplizierter. Alles Wissenswerte darüber erfahren Sie hier!

Genug von Zahlen. Wir wollen uns nun ansehen, wie die Maschine sie knackt. Dazu müssen wir Bescheid wissen über die intime Struktur des Prozessors – über seine *Architektur.*

Der Z8Ø-Mikroprozessor ist das Ergebnis von rund fünfundzwanzig Jahren Computerentwicklung und ein recht modernes Ding, für den Anfänger also vielleicht nicht der ideale Einstieg. Ich möchte deshalb einen einfachen Prozessor beschreiben, der Ende der vierziger Jahre gebaut worden sein könnte (aber nicht wurde), nur um die wichtigen Grundbegriffe darzustellen, die für praktisch alle heutigen Geräte gelten, ohne mir Gedanken über die Schnörkel machen zu müssen, mit denen wir uns später befassen können (ab Kapitel 7).

Wir wollen davon ausgehen, daß unsere erfundene Maschine einen Speicher von 16 Bit-Wörtern und eine Anzahl von 16 Bit-Spezialregistern besitzt, die Sie unten sehen:



Sehen wir uns zuerst den Speicher an. In BASIC hätten wir jeden dieser Speicherplätze nennen können, wie wir wollten, aber so entgegenkommend ist unser erfundener Prozessor nicht. Er besteht darauf, jeden Platz auf eindeutig feststehende Weise zu numerieren, wie ich es angezeigt habe, beginnend bei Null.

Diese Zahlen nennt man *Speicheradressen,* und ich habe sie in Hex numeriert, obwohl Sie immer daran denken sollten, daß die Codierung letzten Endes binär erfolgt.

Was kann in einem Speicherwort festgehalten werden? Jede Anordnung von 16 Bits. Liegt nahe, aber worauf ich hinauswill, ist, daß diese 16 Bits bedeuten können, was wir wünschen. Wenn wir möchten, daß sie eine ganze Zahl, codiert als Zweierkomplement, bedeuten sollen, enthält ein Wort eine Zahl im Bereich −32768 bis 32767. Sollen sie eine positive ganze Zahl ohne Vorzeichenbit bedeuten, liegt die Zahl im Bereich Ø bis 65535. Wir können das Wort auch in zwei Felder von je 8 Bits aufspalten, von denen jedes ein alphabetisches, Interpunktions- und Grafiksymbol bedeuten kann. Wie Tweedledee in "Alice im Wunderland" (oder war es Tweedledum?) sagte: "Wenn *ich* ein Wort gebrauche, bedeutet es genau das, was es für mich bedeuten soll – nicht mehr und nicht weniger." Manchmal meint man, Lewis Carroll sei seiner Zeit voraus gewesen.

Nun zu den Spezialregistern. Für den Anfang nur das A-Register. Es wird jedesmal dann verwendet, wenn Sie Arithmetik betreiben, also rechnen. Das Resultat jeder Rechnung, die Sie von der Maschine verlangen, wird ins A-Register gestellt. (Manchmal sagt man übrigens *Akkumulator* dazu.) Die meisten arithmetischen Operationen arbeiten mit zwei Werten; es hat keinen Sinn, der Maschine zu befehlen, sie solle 3+ ausrechnen – Sie müssen ihr schon sagen, wozu 3 addiert werden soll. Einer dieser Werte muß im A-Register stehen, bevor die Addition ausgeführt wird. Sie können also einen Befehl schreiben wie

ADD (1A3)

Der Computer faßt das so auf:

1 Addiere den Inhalt von Speicherplatz 1A3 zum Inhalt des A-Registers. (Die Klammern um 1A3 sollen anzeigen, daß der *Inhalt* von 1A3 und nicht die *Zahl* 1A3 addiert werden soll.)
2 Stell das Resultat ins A-Register zurück.

Wir haben eben unseren ersten Befehl auf Maschinenebene geschrieben. Echter Maschinencode ist das zwar noch nicht, kommt aber schon nah heran. Sehen wir uns die allgemeine Form an. Sie besteht aus einem Operationscode ADD und einer Adresse (1A3). So werden viele Befehle aussehen. Übrigens ist das Leben auch zu kurz, um dauernd "Operationscode" zu sagen; man verwendet den Ausdruck "Opcodes".

Ein Additionsprogramm

Befassen wir uns mit einer Folge von Maschinenbefehlen, die der BASIC-Anweisung

LET R = B + C

entsprechen würden. Als erstes müßten wir R, B und C konkrete Adressen zuteilen. Nehmen wir an, daß sie der Reihe nach 1Ø3, 1Ø4 und 1Ø5 sind. Wir müssen den Inhalt von 1Ø4 ins A-Register stellen. Erfinden wir dafür LD (für load accumulator = Lade Akkumulator):

LD (1Ø4)

und addieren wir den Inhalt von 1Ø5

ADD (1Ø5)

Schließlich brauchen wir noch einen Weg, den Inhalt des A-Registers nach 1Ø3 zurückzuspeichern. Wir erfinden also einen "Speicher"-Befehl:

ST (1Ø3)

Jetzt haben wir ein einfaches Programm auf Maschinenebene, das aus drei Befehlen besteht:

```
LD (1Ø4)      B in A-Register laden
ADD (1Ø5)     C addieren
ST (1Ø3)      Ergebnis nach R stellen
```

Wie bekommen wir die Maschine dazu, ein solches Programm zu fahren?

Wir sind daran gewöhnt, daß ein Programm in der Maschine gespeichert wird, *bevor* es zur Ausführung kommt. Schließlich wären Sie, wenn Sie die BASIS-Anweisung schreiben:

```
1Ø PRINT "HALLO WELT"
```

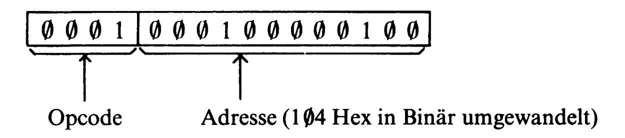
doch einigermaßen verblüfft, wenn sofort, nachdem Sie ENTER gedrückt haben, die MELDUNG "HALLO WELT" erschiene. Sie gehen davon aus, daß sie festgehalten wird, bis Sie sie brauchen. Für ein Programm auf Maschinenebene gilt dasselbe. Wo könnte ein Befehl auf natürlichere Weise gespeichert werden als in einem Speicherwort? (Ein Wort soll die Bedeutung haben, die Sie wünschen – wissen Sie noch?) Das setzt natürlich voraus, daß die Opcodes LD, ADD und so weiter als Bitanordnungen codiert werden, aber dazu brauchen wir nur eine Tabelle von Bitanordnungen zu erfinden, die dann so aussähe:

Opcode-Mnemonik	Binärcode
ADD	ØØØØ
LD	ØØØ1
ST	ØØ1Ø

Jedesmal, wenn uns ein neuer Opcode einfällt, den wir brauchen, fügen wir ihn der Tabelle an.

Ich bin oben davon ausgegangen, daß alle Opcodes einen Binärcode von 4 Bits haben. Das läßt 16 verschiedene Anordnungen und damit 16 verschiedene Befehle zu. Nach modernen Maßstäben ist das ein kleiner Befehlsvorrat, für unseren erfundenen Spielzeugcomputer reicht er aber aus. Wir haben für das ganze Wort 16 Bits, also bleiben 12 für den Adreßteil des Befehls.

LD (1Ø4) sieht innerhalb der Maschine dann so aus:



Wenn Sie eine Bitanordnung gesehen haben, kennen Sie alle. Von jetzt an schreiben wir also die Hexversionen von Befehlen. Das ist um eine Spur weniger mühsam.

Der Programmzähler

Nehmen wir an, wir speichern unser Programm aus 3 Befehlen ab Platz ØFF:

Speicher	Adresse
	ØFE
11Ø4	ØFF
Ø1Ø5	1ØØ
21Ø3	1Ø1
	1Ø2
	1Ø3
	1Ø4
	1Ø5
	1Ø6

Nun brauchen wir eine Methode, dem Computer zu sagen: "Bring das Ganze ins Rollen, indem du den Befehl in ØFF ausführst, dann den in 1ØØ, dann einen in 1Ø1." Dazu ist das PC-Register, der *Programmzähler*, da. Er dient als eine Art Lesezeichen für den Computer. Wir fahren das Programm, indem wir den PC auf die Adresse des ersten Befehls initialisieren. Während die Maschine diesem Befehl gehorcht, wird der PC automatisch um 1 akutalisiert, so daß das System, wenn es zurückspringt, um den PC zuuntersuchen, hergeht und den nächsten Befehl ausführt, und so weiter.

Das hat aber einen Haken. Während der letzte Befehl (in 1Ø1) zur Ausführung kommt, wird der PC wie gewohnt um 1 erhöht, und sobald die Maschine wieder hinsieht, findet sie 1Ø2, springt also dorthin, um den dortigen Befehl

auszuführen. Was für einen Befehl? In 1Ø2 haben wir keinen hineingesetzt. Ah! In 1Ø2 muß aber von einem vorherigen Programm noch ein Befehl stehen oder, als der Computer eingeschaltet wurde, einer hineingesetzt worden sein. Die Maschine wird dieses Muster also wie einen Befehl auslegen, weil wir das von ihr verlangt haben. Dann wird sie weitergehen zu den Plätzen 1Ø3, 1Ø4 und 1Ø5; dort speichern wir aber Daten! Wenn die Zahl in 1Ø4 also beispielsweise 2ØFF ist, wird der Computer das auslegen als:

ST (ØFF)

was den Inhalt des A-Registers nach ØFF kopiert und damit den ersten Befehl unseres Programms zerstört! Offenkundig brauchen wir einen "Halt"-Befehl (ich verwende das mnemotechnische HLT), der das Aktualisieren des PC zum Stillstand bringt. Das Programm lautet jetzt also:

LD (1Ø4)

ADD (1Ø5)

ST (1Ø3)

HLT

Ein wichtiger Punkt. Genau deshalb, *weil* wir Wörter verwenden, die bei verschiedenen Gelegenheiten Verschiedenes bedeuten, müssen wir ein besonderes Auge auf die Folgerungen haben, die der Computer aus dem ziehen wird, was wir ihm auftragen. Wenn wir mit ADD verlangen, er soll den Inhalt eines Speicherplatzes ins A-Register addieren, geht er davon aus, daß der Speicherplatz eine Zahl enthält. Er prüft das nicht nach, weil er es gar nicht kann – jede Bit-Anordnung könnte eine Zahl darstellen. Ebenso könnte jede Bit-Anordnung einen Befehl darstellen; wenn also der PC auf einen Speicherplatz zeigt, wird dessen Inhalt als Befehl ausgeführt.

Die Regel lautet: *Halten Sie Daten und Programme klar getrennt.* Wenn Sie es nicht tun, müssen Sie damit rechnen, in regelmäßigen Abständen vor unlösbare Rätsel gestellt zu werden! Wie ich schon angedeutet habe, kann ein ganzes Programm während des Laufs spurlos verschwinden!

5 Sprünge und Subroutinen

Weitere Befehle: Die Funktionen von Programmzähler und Stapel.

Bis jetzt sieht unser Befehlsvorrat noch ein wenig mager aus. Wir haben LD und ST, die im Speicher etwas bewegen, ADD, eigentlich recht primitives Rechnen, und wir können die Dinge mit HLT zum Stehen bringen.

Wir päppeln die Rechnerfähigkeiten ein bißchen auf mit SUB, was den Inhalt eines Platzes vom A-Register abzieht, das ist aber auch schon alles, was wir bekommen. Kein Multiplizieren, kein Dividieren, schon gar kein Wurzelziehen.

Was wir wirklich brauchen, ist ein Vorrat an Verzweigungsbefehlen entsprechend dem IF . . . THEN . . . in BASIC!

Sprünge

Aus der üblichen Reihenfolge zu einem Befehl abzuzweigen, ist recht leicht. Wir müssen nur den Inhalt des PC verändern. Dazu verwenden wir also einen Befehl wie

JP 416 spring zu 416

So oft er ausgeführt wird, setzt er 416 in den PC. Dem System wird "vorgemacht", der nächste Befehl sei in 416 enthalten, und es geht weiter zu 417, 418, usw., bis der nächste "Sprung"-Befehl auftaucht. Dem JP-Code kann natürlich jede Adresse folgen.

Dieser Befehl gleicht eher einem GO TO als einem IF . . . THEN . . . Was wir nötig haben, ist ein Befehl, der den PC nur dann neu setzt, sobald irgendeine Bedingung erfüllt wird. Die einfachste Prüfung ist die, ob das A-Register Null enthält.

JPZ 2A7 spring zu 2A7 nur, wenn A-Reg. Ø enthält

Oder ein anderer:

JPN 14E spring zu 14E nur, wenn Inhalt A-Reg. negativ

Das ist das Mindeste, was wir uns leisten dürfen, weil wir jetzt auf eine positive (nicht Null betragende) Zahl prüfen können, indem wir darauf achten, wann das Programm *weder* zu JPZ- noch zu JPN-Befehlen springt . . .

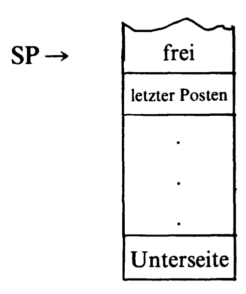
Subroutinen und Stapel

Weil wir schon beim Thema sind: Wenn innerhalb des Programms die Steuerung von einem Platz zum anderen übertragen werden soll, warum nicht etwas in der Art von GO SUB und RETURN bei BASIC?

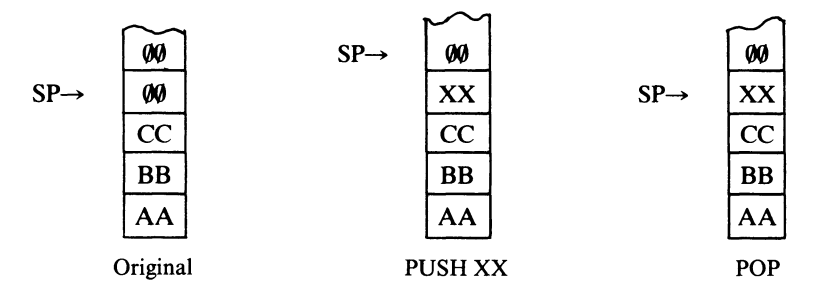
Dann hätten wir einen Befehl:

CALL 2Ø5 ruf die Subroutine ab 2Ø5 auf

Was bewirkt das? Offenkundig wird dadurch 2Ø5 in den PC gesetzt, aber dafür könnten wir auch ein JP verwenden. CALL erfüllt eine zweite Funktion: Es speichert die Adresse des Befehls nach CALL, so daß, wenn ein "Return" (Opcode: RET) erscheint, es die gespeicherte Adresse in den PC zurückladen kann, um das Hauptprogramm dort fortzusetzen, wo es aufgehört hatte. Das wird durch einen *Stapel* bewirkt. Ein Stapel ist ein Speichersegment mit feststehender "Unterseite" und variabler Oberseite. Ein Stapelzähler SP enthält die Adresse der Oberseite. Man nennt das "Zeiger", weil er auf die Oberseite des Stapels zeigt, nämlich so:



Zusätzliche Größen können dadurch auf den Stapel geschoben werden, daß man SP eins hinaufschiebt und die neue Größe in den Speicher setzt; mit POP können Sie vom Stapel genommen werden, indem man SP um eins verringert. (Man braucht die entfernte Größe aus dem Speicher nicht zu löschen, weil die Stapelroutinen alles unbeachtet lassen, was oberhalb des SP vorgeht.) Beispiel:



Der SP zeigt in Wirklichkeit auf den ersten *unbenutzten* Platz. (Außerdem zeigt der Z8Ø-Maschinenstapel im Spectrum nach unten, nicht nach oben. Zerbrechen Sie sich darüber aber nicht den Kopf; für den Anwender, der die eigentlichen Stapeladressen nicht zu kennen braucht, spielt das keine Rolle.)

Stapel funktionieren nach dem Grundsatz: "Zuletzt rein, zuerst raus." Stellen Sie sich einen Stapel Bücher auf einem Schreibtisch vor. PUSH heißt soviel wie: "Leg ein Buch oben auf den Stapel" und POP (praktisch): "Nimm

ein Buch vom Stapel herunter." Wenn Sie also der Reihe nach drei Größen X, Y und Z mit PUSH bewegen

PUSH X

PUSH Y

PUSH Z

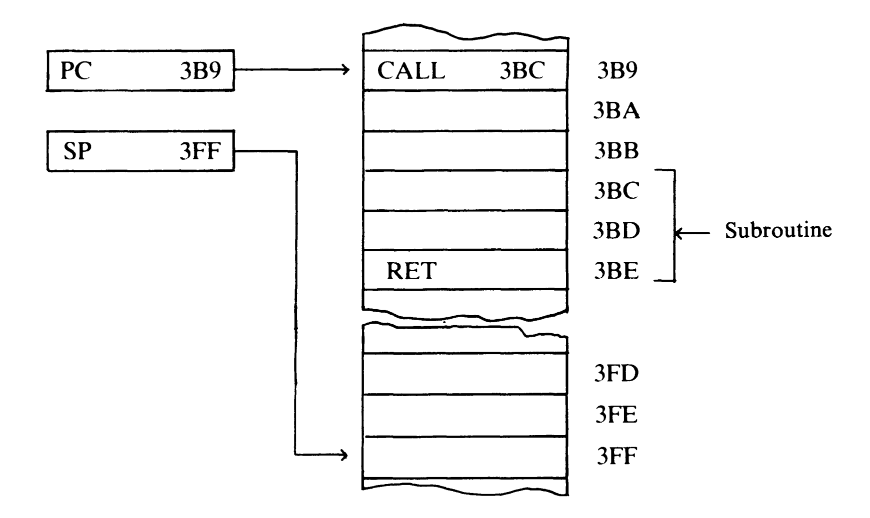
dann müssen Sie, um sie in der richtigen Reihenfolge herunterzunehmen, verwenden:

POP Z

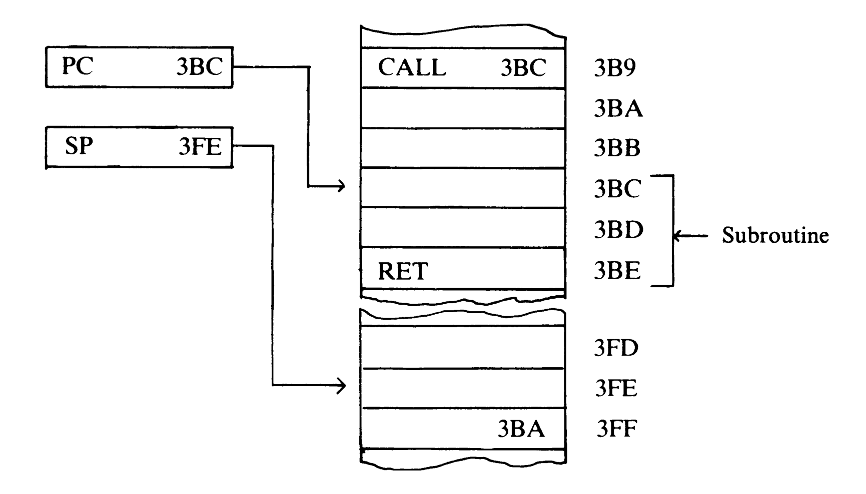
POP Y

POP X

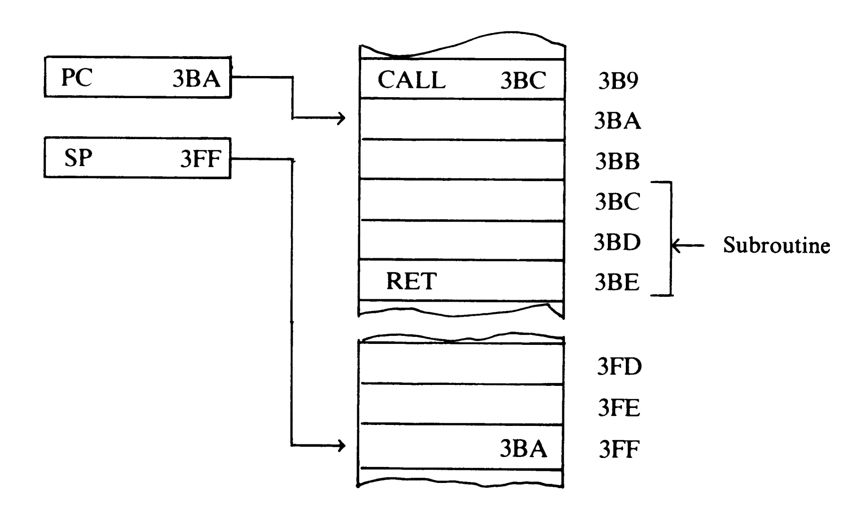
Eine Subroutine verwendet den Stapel dazu, sich seine Returnadresse zu merken. Bei Ausführung eines CALL wird die Returnadresse (die Adresse von CALL + 1) auf den Stapel geschoben. Erscheint RET, wird der Stapel in den PC gepoppt. Hier ein Beispiel:



Das CALL wird nun ausgeführt:



Es ist befolgt worden, die Returnadresse befindet sich auf dem Stapel. Das Programm geht die Subroutine durch, bis es das RET erreicht, worauf:



und die Steuerung ist wieder im Hauptprogramm.

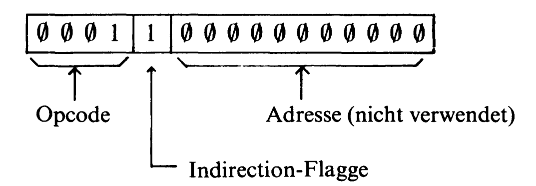
6 Indirekte Steuerung und Indizieren

Der Privatdetektiv auf der Fährte des Opfers: Wie man den Inhalt einer Adresse nutzt, um auf eine andere zu zeigen.

Zu besprechen sind nun nur noch zwei Register. Beide haben ähnliche Funktionen. Sie können den Adreßteil eines Befehls verändern, während das Programm läuft.

Indirekte Steuerung

Sehen wir uns zuerst an, wie das I-Regiser damit zurechtkommt. Wir erfinden einen neuen Opcode LDI oder "lade indirekt". Wie bei HLT ist damit keine Adresse verbunden. Für die Maschine entspricht das ganz einem LD, nur wird das hochwertige Bit des Adreßfeldes auf "1" gesetzt. Dieses Bit nennt man *Indirection-Flagge*. Es zeigt der Maschine lediglich an, daß indirekte Steuerung in Kraft ist. Die binäre Form des LDI-Befehls ist somit:



Der Hexcode ist 18ØØ. Stößt die Maschine auf diesen Befehl, so verwendet sie die im I-Register vorhandene Zahl als die effektive Adresse. Enthält das I-Register also 1E4, und es wird ein LDI-Befehl ausgeführt, ist die Wirkung genau dieselbe, als hätte der Befehl LD 1E4 gelautet. Mit anderen Worten: Das I-Register dient als Speicherzeiger. Den können wir nach Herzenslust umherbewegen, falls wir damit zu rechnen vermögen. Das bedeutet, Werte in das A-Register setzen, weil das der einzige Ort ist, wo wir rechnen können. Wir erfinden also einen Opcode XAI für "tausche Inhalt des A-Registers gegen Inhalt des I-Registers".

Selbstverständlich kann die Flagge für indirekte Steuerung bei jedem Befehl gesetzt werden, der einen Adreßteil besitzt. Wir können also STI, JPI, ADDI usw. verwenden; in jedem Fall lauten die letzten drei Ziffern des Hexcodes 8ØØ.

Ein Beispiel

Sehen wir uns ein Beispiel an, bei dem diese Ideen genutzt werden. Nehmen wir an, wir wollten ein 1D-Array von Länge 2Ø initialisieren, das die Zahlen 2, 4, 6, 8 . . . 4Ø enthalten soll. Anders ausgedrückt: Wir wollen eine Maschinencode-Entsprechung für den BASIC-Code:

```
FOR c = 1 TO 2Ø
LET a (c) = c * 2
NEXT c
```

Es gibt eine Reihe von Werten, die irgendwo im Speicher stehen müssen, damit das klappt. Sie sind 1 (weil der Schleifenzähler jeweils um 1 erhöht wird), 2 (weil das die Inkrementierung für den Arrayinhalt ist) und 2Ø (als Test für das Schleifenende). Im Augenblick will ich mich nicht darum kümmern, wo genau diese Zahlen gespeichert werden sollen, also lasse ich zu, daß Adressen vorübergehend mit Namen bezeichnet werden (genau wie bei BASIC-Namen). Wir müssen diese freilich in Zahlen verwandeln, wenn wir schließlich zum Maschinencode kommen. Das ist ein Fall des Ersten Jonesschen Computergesetzes: "Verschiebe nie auf morgen, was du übermorgen kannst besorgen." Wir wollen also unterstellen, daß die Zahlen, die wir brauchen, an den Plätzen N1, N2 und N2Ø verfügbar sind. Ebenso haben wir einen Platz namens BASE, der die Adresse des ersten Arraylements enthält, und einen namens COUNT, der als Schleifenzähler dient.

Als erstes setzen wir das I-Register so, daß es auf die Unterseite des Arrays zeigt:

```
LD      BASE
XAI
```

Dann setzen wir COUNT auf 1:

```
LD      N1
ST      COUNT
```

Das verdoppeln wir nun (indem wir es ins A-Register zurückdatieren) und speichern es an dem Platz, auf den das I-Register zeigt. (Wir sprechen kurz vom "Speichern *durch* das I-Register".)

```
ADD     COUNT
STI
```

Wir "entdoppeln" den Wert auf dem A-Register wieder, ziehen 2Ø ab und prüfen, ob das Ergebnis Null ist. Wenn ja, sind wir fertig:

```
SUB     COUNT
SUB     N2Ø
JPZ     OUT
```

OUT ist eine weitere, noch unbestimmte, Adresse. Wir wissen noch nicht, wo sie ist, weil wir nicht wissen, wo das Programm aufhört, also ist es wieder nützlich, ihr vorübergehend einen Namen zu geben.

Wenn die Verzweigung nicht stattfindet, fügen wir 1 zu COUNT hinzu:

```
LD      COUNT
ADD     N1
ST      COUNT
```

und inkrementieren das I-Register um 1:

```
XAI
ADD     N1
XAI
```

Das laufende COUNT ist nun wieder im A-Register, so daß wir mit einer Schleife zur Verdoppelungsoperation zurückkehren können:

```
JP      LOOP
```

vorausgesetzt, wir geben dem "ADD COUNT-Befehl die symbolische Adresse "LOOP". Das geschieht, indem wir dem Befehl seine symbolische Adresse voranstellen, gefolgt von einem Doppelpunkt:

```
LOOP:   ADD    COUNT
```

Das Gleiche können wir tun, um die benötigten Anfangswerte zu setzen, indem wir einen neuen Opcode HEX definieren, der einfach ein Wort auf einen verlangten Wert setzt. Eigentlich ist das gar kein Opcode, weil er keinem Maschinenbefehl entspricht. Wir reden deshalb von einer Pseudo-Operation. Das ganze Programm sieht so aus (die Zahlen in der Spalte ganz links und ganz rechts beachten Sie vorerst noch nicht):

```
Ø2Ø             LD      BASE     1 Ø33
Ø21             XAI              A ØØØ
Ø22             LD      N1       1 Ø3Ø
Ø23             ST      COUNT    2 Ø32
Ø24  LOOP:      ADD     COUNT    Ø Ø32
Ø25             STI              2 8ØØ
Ø26             SUB     COUNT    4 Ø32
Ø27             SUB     N2Ø      4 Ø31
Ø28             JPZ     OUT      6 Ø47
Ø29             LD      COUNT    1 Ø32
```

```
Ø2A          ADD   N1      Ø Ø3Ø
Ø2B          ST    COUNT   2 Ø32
Ø2C          XAI           A ØØØ
Ø2D          ADD   N1      Ø Ø3Ø
Ø2E          XAI           A ØØØ
Ø2F          JP    LOOP    5 Ø24
Ø3Ø N1:      HEX   ØØØ1    Ø ØØ1
Ø31 N2Ø:     HEX   ØØ14    Ø Ø14
Ø32 COUNT:   HEX   ØØØØ    Ø ØØØ
Ø33 BASE     HEX   ØØØØ    Ø ØØØ
```

Die einzige symbolische Adresse, die in der linken Spalte nicht erscheint und deshalb noch unspezifiert bleibt, ist OUT. Darüber zerbrechen wir uns später den Kopf.

Die Form des Programms, das wir jetzt geschrieben haben, wird *Assemblercode* genannt. Bei modernen, leistungsstarken Computern gibt es ein *Assemblerprogramm*, dessen Aufgabe darin besteht, das für uns in echten Maschinencode zu übersetzen.

Assemblieren von Hand

Leider besitzen weder unsere erfundene Maschine noch der Spectrum ein solches Programm (man kann es allerdings im Handel erwerben). Wir müssen das also selbst machen. Wir brauchen eine Tabelle von Opcodes und ihren entsprechenden Hexwerten:

Opcode	Hex
ADD	Ø
LD	1
ST	2
HLT	3
SUB	4
JP	5
JPZ	6
JPN	7
CALL	8
RET	9
XAI	A

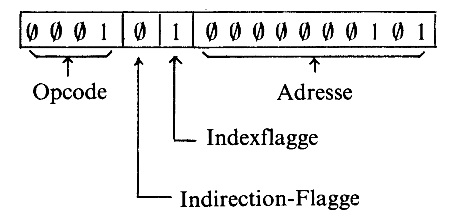
Außerdem müssen wir wissen, wo der Programmbeginn ist. Das ist eine mehr oder weniger willkürliche Entscheidung, also wollen wir unterstellen, er sei bei Ø2Ø. Da jeder Befehl ein Wort besetzt, können wir die Adresse jedes Befehls schriftlich festhalten. Man kann sehen, daß ich das auf der linken Seite des Programms auf Seite 34 getan habe. Wir können die Opcodes und Adressen nun durch ihre Hexwerte ersetzen. Beispielsweise wird LD BASE zu 1Ø33, weil BASE nun als Ø33 identifiziert ist. Die rechte Spalte auf Seite 34 zeigt den vollständigen Code.

Der einzige Befehl, der noch einen Kommentar nötig hat, ist JPZ OUT, codiert als 6Ø47. Weshalb steht OUT gerade bei Ø47? Es könnte auch anderswo sein, aber Ø47 ist die erste Position, wo es sein *kann.* Der Grund: Das Array nimmt den Platz von Ø33 bis Ø46 ein (zwanzig Wörter), und wir wollen ja nicht im Datenbereich des Programms herumtappen.

Das Index-Register

Bei Verwendung des X-Registers wird die echte Befehlsadresse dadurch gebildet, daß man das Adreßfeld dem Inhalt des X-Registers hinzuaddiert. Beispiel: Wenn das X-Register 4ØØ enthält, hat der Befehl LDX ØØ5 dieselbe Wirkung wie LD 4Ø5.

Wir stehlen uns noch ein Bit des Adreßfelds, um anzuzeigen, wann Indizieren im Gange ist, also sieht der Befehl LDX ØØ5 folgendermaßen aus:



In Hex ist das 4Ø5. Eigentlich gibt es beim Indizieren nichts, was Sie nicht auch mit indirekter Steuerung leisten könnten. Indizieren bewirkt lediglich, daß Rechnen mit Adressen automatisch ausgeführt und die Arbeit nicht Ihnen überlassen wird.

7 Endlich: Der Z80

Die echte Architektur des Z8Ø-Mikroprozessors, Herz (oder Hirn?) Ihres Spectrums.

Tut mir leid, daß Sie sich durch die letzten zehn Seiten, oder wieviele es waren, hindurchquälen mußten, ohne irgend etwas ausprobieren zu können, aber wenn Sie wirklich begriffen haben, worum es ging, werden Sie feststellen, daß es eine Kleinigkeit ist, den Z8Ø zu verstehen.

Bevor wir uns mit der Architektur des Z8Ø befassen (so ganz genau stimmt die Kapitelüberschrift ja nun doch nicht), nehmen wir uns ein paar von den Problemen des Prozesses vor, die ich eben angesprochen habe.

Erstens läßt der Operationscode mit seinen 4 Bits nur 16 verschiedene Befehle zu. (Na gut, ein bißchen haben wir geschwindelt, als wir zuließen, daß die Flaggen der indirekten Steuerung und des Indizierens in das Adreßfeld hinüberreichen durften, was aber andererseits wieder bedeutet, daß wir die Adreßgröße und damit die Höchstgröße des Speichers begrenzt haben!) Der Z8Ø hat 694 Befehle! Jedem einzelnen eine eigene Bitanordnung zuzugestehen, heißt, daß wir ein Feld von 8 Bits (ein Byte) brauchen; und sogar dann muß man ein bißchen schwindeln.

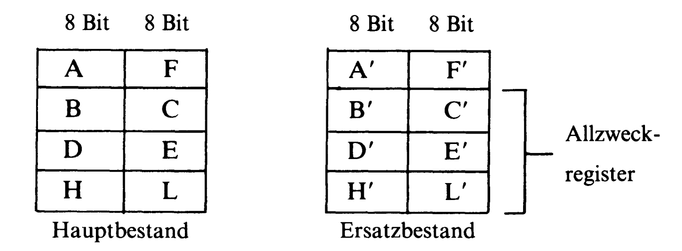
Zweitens verwendet unsere erfundene Maschine den Speicher recht sorglos. Manche Adressen (etwa HLT, LDI, STI) benützen das Adreßfeld nicht, so daß eine Folge solcher Befehle in jedem Wort 1Ø Bits vergeudet. Der Z8Ø löst dieses Problem dadurch, daß er verschiedenen Befehlen verschiedene Längen zugesteht. Manche Befehle haben kein Adreßfeld und sind genau 1 Byte lang. Andere haben ein Adreßfeld von 1 Byte Länge und sind damit 2 Bytes lang. Andere, insgesamt 3 Byte lang, haben ein Adreßfeld von 2 Bytes Länge. Es gibt sogar welche, die Opcodes von 2 Bytes besitzen! Das bedeutet, daß der PC nicht für jeden ausgeführten Befehl um 1 inkrementieren kann. Er muß inkrementieren um die Länge des Befehls.

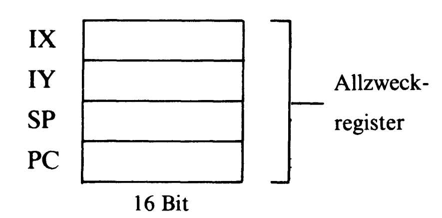
Drittens müssen wir stets Wörter von 16 Bits verarbeiten, was nicht günstig ist, wenn wir uns mit Zeichen befassen, die in der Regel jeweils ein Byte besetzen. Es wäre also schön, wenn 8 Bit- *und* 16 Bit-Operationen zulässig wären.

Viertens kann ärgerlich sein, daß es nur ein Allzweckregister (das A-Register) gibt. Oft heißt das, daß Zwischenergebnisse vorübergehend in den Speicher zurückgestellt werden müssen, während andere Berechnungen stattfinden. Der Z8Ø hat eine Reihe von Allzweckregistern, obschon, wie wir sehen werden, die genaue Anzahl davon abhängt, wofür wir sie verwenden.

Die Register

Hier die Registerorganisation:





Den Ersatzbestand wollen wir vorerst nicht beachten.

Die Register treten paarweise auf, was darauf hindeutet, daß sie entweder als 8- oder als 16 Bit-Register verwendet werden können. Beispielsweise können wir uns auf das B-Register (8 Bits) oder das BC-Register (16 Bits) beziehen. Die Register B, C, D, E, H und L können alle auf diese Weise genutzt werden (*ausschließlich* als Paare BC, DE und HL), aber die Register A und F sind nichts anderes als 8 Bit-Register und können *nicht* kombiniert werden. Bei den 16 Bit-Paaren ist, wie zu erwarten, das höherwertige Bit das linke (B, D, H).

Es gibt zwei Index-Register, IX und IY, einen Stapelzeiger (SP) und einen Programmzeiger (PC). Wie, keine indirekte Steuerung? In der Tat kann jedes der 16 Bit-Allzweckregisterpaare (BC, DE oder HL) für indirekte Steuerung verwendet werden; der Einfachheit halber wollen wir zu diesem Zweck aber stets das HL-Register verwenden.

Es gibt zwei Befehlssätze, einen für den Umgang mit 8 Bit-Operationen, den anderen für den mit 16 Bit-Operationen. Wir fangen an mit den "Lade"-Befehlen von 8 Bits.

8 Adressierarten und die LD-Befehle

Es gibt fünf verschiedene Wege, sich auf die in Z8Ø-Befehlen verwendete Adresse zu beziehen. Am leichtesten versteht man sie unter Bezug auf die LD-Befehlsgruppe, die Daten von einem Platz zum anderen verschieben.

Sehen wir uns die "Lade"-Operation (LD) als ein Beispiel der 8 Bit-Gruppe an. Sie ähnelt sehr dem LD-Befehl bei unserer erfundenen Maschine, nur sind zwei Adressierarten zugelassen: *Register zu Register* und *sofort*. Das sind insgesamt fünf, mit *direkt, indirekt* und *indiziert* wie vorher verfügbar.

1 Direktes *Adressieren*
Das sieht ganz ähnlich aus wie unsere erfundene Entsprechung; wir müssen nur, weil mehr als ein Register vorhanden ist, angeben, welches Register geladen werden soll:

LD A, (ØF1C)

Das lädt den Inhalt von ØF1C in das A-Register. Beachten Sie, daß die Bewegung üblicherweise von rechts nach links geht, so daß wir schreiben können:

LD (ØF1C), A

womit gemeint ist "kopiere den Inhalt des A-Registers nach ØF1C". (In Wahrheit ist das A-Register das einzige Register mit 8 Bits, das direkt adressiert werden kann.)

2 *Indirektes Adressieren*
Auch das ist ganz normal. Da wir uns für indirekte Steuerung auf HL einigen wollen, lautet das Befehlsformat:

LD A, (HL)

Das bedeutet: "Lade das A-Register *durch* das (das heißt, von der Adresse enthalten in) HL-Register." Um Daten in die andere Richtung weiterzugeben, könnten wir verwenden:

LD (HL), A

was den Inhalt von A an die in HL *enthaltene* Adresse setzt. (Für diesen Befehl sind auch andere Register als A zulässig.)

3 *Indiziertes Adressieren*
Hier müssen wir angeben, welches Indexregister in Gebrauch ist, und den Umfang der Versetzung:

LD A, (IX + 2E)

Beachten Sie, daß ich bei direktem Adressieren eine Adresse von 4 Hexziffern angegeben habe, weil für die Adresse 16 Bits (4 Bytes) zulässig sind. Der Versetzungswert in einem Befehl mit indizierter Adresse muß jedoch in 1 Byte enthalten sein, so daß ich nur zwei Hexstellen verwendet habe.

4 *Register zu Register*
Zwischen Registern können wir Daten so übertragen:

LD D, B

was bedeutet: "Lade den Inhalt von B in D."

5 *Sofort*
Hier werden statt der Datenadresse Daten selbst in das Adreßfeld geladen. Wir können also schreiben:

LD B, Ø7

mit der Bedeutung: "Stell die Zahl 7 nach B." Beachten Sie auch hier, daß die Zahl zwei Hexstellen umfaßt, weil sie im Einzelbyte des B-Registers gespeichert werden muß. Beachten Sie ferner, daß ein "LD" in Wirklichkeit ein *Kopieren* ist; die Zahlen werden in ihren ursprünglichen Adressen oder Registern aufbewahrt, am Ziel jedoch eine Kopie abgelegt.

Hexcodes

Sehen wir uns an, wie jeder dieser Befehle in Hex aussieht. Eine komplette Aufstellung finden Sie in Anhang 6 (siehe Beilagekarte).

1 LD A, (Ø1FC)

Zuerst schlagen wir den Opcode für den LD A, (nn)-Befehl nach. (Das nn steht für eine beliebige 2 Byte-Adresse). Das ist 3A. Man möchte also annehmen, der Befehl sei zu codieren als:

3A ØF1C

Leider gibt es eine kleine Komplikation, hervorgerufen durch die Art, wie der Z8Ø über Zahlen denkt; er will das am wenigsten bedeutsame (jüngere) Byte einer Adresse zuerst haben. Wir müssen die *Adreßbytes also vertauschen:*

3A 1CØF

Das ist ein wenig ärgerlich, aber man gewöhnt sich bald daran. Das ist eine unabdingbare Regel für 2 Byte-Zahlen in Z8Ø-Befehlen: zuerst das *jüngere Byte, dann das ältere:* daher die vielen PEEK X + 256* PEEK (X + 1) im Sinclair-Handbuch.

Der Befehl LD (nn), A hat Code 32, also wird aus

LD (Ø1FC) dann 32 1CØF

2 LD A, (HL)

Das ist einfach. Es gibt keinen Adreßteil, also ist das nur ein 1 Bit-Opcode. Schlagen Sie nach, Sie finden 7E.
Ähnlich wird LD (HL), A als 77 codiert.

3 LD A, (IX + 2E)

Der Befehl lautet allgemein LD A, (IX + d), wobei d eine 2 Byte-Verschiebung (in Zweierkomplementschreibung) anzeigt, und der Code lautet DD 7E. (Beachten Sie, daß ist ein 2 Byte-Opcode!) Der Befehl lautet dann:

DD 7E 2E

wo das Byte 2E die in diesem Fall gewählte Verschiebung ist.

4 LD D, B

Auch hier kein Problem. Der Code ist 5Ø.

5 LD B, Ø7

Der Opcode ist Ø6, der Befehl also Ø6 Ø7.

9 Maschinencode speichern, fahren und sichern

Wir müssen nicht nur lernen, Maschinencode zu schreiben, sondern auch, den Spectrum dazu zu bringen, daß er ihn ausführt, und wie wir unsere Bemühungen für die Nachwelt erhalten wollen.

Das Hauptziel dieses Kapitels besteht darin, ein einfaches BASIC-Nutzprogramm zu entwickeln, das dem Maschinencode etwas von seiner Mühsal nimmt. Eine verfeinerte Version davon steht in Anhang 7.

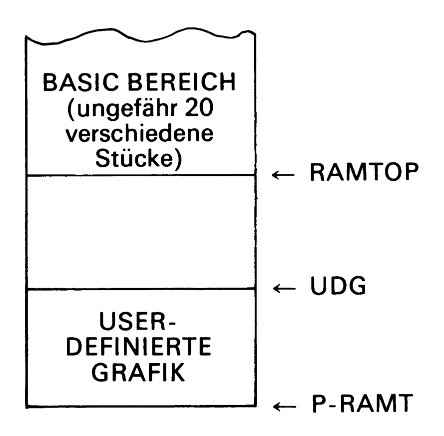
Fangen wir an mit dem Spectrum-Speicher. Es gibt ihn in zwei Arten:

Typ	Startadresse		Schlußadresse	
	Dezimal	Hex	Dezimal	Hex
ROM	Ø	Ø	16383	3FFF
RAM	16384	4ØØØ	32767	7FFF (16K-Gerät)
			65535	FFFF (48K-Gerät)

Das ROM enthält das Betriebssystem, den BASIC-Interpreter und so weiter; es wäre katastrophal, wenn wir daran herumdoktern würden, und man hat es so eingerichtet, daß das nicht geht. ROM heißt "Read Only Memory" – Nur-Lese-Speicher. Man kann mit PEEK hinein, aber nicht mit POKE.

Mit dem RAM (Random Access Memory – Speicher mit wahlfreiem Zugriff) können wir machen, was wir wollen, und im RAM muß der Maschinencode gespeichert und ausgeführt werden. Ein mögliches Problem ist, daß das BASIC-System den RAM ebenfalls benützen will. Wenn wir den Maschinencode an beliebiger Stelle speichern, löscht ihn das BASIC-System einfach, überschreibt ihn oder macht ihn sonst kaputt. Wir müssen den Maschinencode dort festhalten, wo er vom BASIC-System nicht benützt (oder wenigstens nicht verändert) wird.

Der zivilisierte Platz (andere sind möglich, siehe weiter hinten im Band) ist oben auf dem Speicher. Hier ein Bild des Speicherbereichs, um den es hier geht:



Die Adressen der Grenzen zwischen diesen Bereichen sind enthalten in drei Systemvariablen, die Adressen von *diesen* bei 16K oder 48K dieselben:

RAMTOP 23730 – 1

UDG 23675 – 6

P-RAMT 23732 – 3

Das sind 2 Byte-Variable, und den Wert von z. B. RAMTOP finden Sie durch Verwendung von

PRINT PEEK 23730 + 256 * PEEK 23731

Schaltet man die Maschine das erstemal ein, werden diese auf Standardwerte gesetzt:

Variable	16K		48K	
	Dezimal	Hex	Dezimal	Hex
RAMTOP	32599	7F57	65367	FF57
UDG	32600	7F58	65368	FF58
P-RAMT	32767	7FFF	65535	FFFF

Das BASIC-System verwendet den Bereich unterhalb und *einschließlich* der Adresse in RAMTOP. Der Teil von UDG bis hinauf zu P-RAMT (Oberseite von RAM physisch) speichert das Format für userdefinierte Grafik. Dieser Bereich kann in der Größe verkleinert oder ganz geräumt werden, wenn man den Wert von UDG verändert; um die Sache aber nicht übermäßig zu komplizieren, gehe

ich davon aus, daß wir den Bereich für userdefinierte Grafik an seinem üblichen Platz lassen wollen.

Wir schaffen Platz für den Maschinencode, indem wir den Wert von RAMTOP herabsetzen. Beispiel: Um 6ØØ Bytes Raum freizumachen und den Maschinencode bei Adresse 32ØØØ beginnen zu lassen, müssen wir RAMTOP verändern zu 32ØØØ – 1 = 31999.

(Das schafft mehr als genug Platz für alles, was in diesem Band vorkommt, und die Startadresse für Maschinencode ist dezimal (32ØØØ) ebenso wie hex (7DØØ) eine runde Zahl, so daß ich nach diesem Wert normiere. Wenn Sie einen anderen Wert wollen, verändern Sie einfach jedesmal 32ØØØ und 31999 oder 7DØØ und 7CFF, wo sie vorkommen, zu dem von Ihnen gewünschten Wert. Damit ich mich mit dem 48K-Gerät nicht eigens beschäftigen muß, verwende ich dafür *ebenfalls* 32ØØØ, aber damit vergeude ich ungefähr 32 K RAM. Sie nehmen vielleicht lieber 64ØØØ, in Hex FAØØ. Während man Maschinencode *lernt,* ist der vergeudete Platz unwichtig, aber bei richtigen Programmen würden Sie RAMTOP ganz gewiß überlegter wählen wollen).

Um RAMTOP herabzusetzen, verwenden Sie folgenden Befehl:

```
CLEAR 31999
```

Das hat folgende Wirkungen:

Es löscht alle Variablen.
Es löscht das Displayfile wie ein CLS.
Es setzt die PLOT-Position auf links unten: (Ø, Ø)
Es führt mit RESTORE den DATA-Zeiger zum Start zurück.
Es löscht den GO SUB-Stapel, der die Rücksprungadressen für die bisher verwendeten Subroutinen enthält.
Es setzt RAMTOP auf 31999 und "sichert" die Adressen 32ØØØ und aufwärts vor Störungen.
Es setzt den neuen GO SUB-Stapel unter diesen neuen Wert von RAMTOP.

Ein Befehl wie

```
CLEAR 31447
```

würde genau dasselbe leisten, RAMTOP aber auf 31447 setzen, wodurch viel mehr Platz bliebe; die Adressen wären von 31448 an aufwärts "ungefährdet".

Am besten verwendet man CLEAR offensichtlich, bevor man irgendwelche Variablen zugeteilt, irgend etwas angezeigt oder irgendeine Subroutine aufgerufen hat. Das Ladeprogramm unten setzt RAMTOP gleich zu Anfang zurück, aber wenn Sie selbst etwas schreiben, ist es keine schlechte Idee, sich den Gebrauch von CLEAR anzugewöhnen, *bevor* man sonst irgend etwas tut.

Warnung

Um das Obige noch einmal hervorzuheben: Damit Sie den Bereich von der Adresse "Adresse" aufwärts sichern, verwenden Sie CLEAR-Adresse –1, *nicht* bloß CLEAR Adresse. Der Wert in einem CLEAR-Befehl ist die *letzte ungesicherte* Adresse, nicht die erste gesicherte. Das ist eine Beschwernis, aber eine kleine. Vergessen Sie sie nicht.

Maschinencode speichern

Was wir brauchen, ist ein Programm, das Hexcodes annimmt und die zugehörigen Bytes geordnet über RAMTOP speichert. Es wird später in diesem Kapitel noch ergänzt, um es vielseitiger zu machen.

```
1Ø  CLEAR 31999
2Ø  PRINT "Basisadresse: 32ØØØ"
3Ø  PRINT "Zahl der Datenbytes: □";
4Ø  INPUT d: PRINT d
5Ø  LET b = 32ØØØ
6Ø  FOR i = Ø TO d
7Ø  POKE b + i, Ø
8Ø  NEXT i
9Ø  LET a = b + d
1ØØ  DIM h$ (2)
11Ø  PRINT "CODE:"
12Ø  INPUT c$
13Ø  IF c$ = "s" THEN GO TO 2ØØ
14Ø  PRINT c$ + "□";
15Ø  LET hs = CODE c$ (1) – 48 – 39 * (c$ (1) > "9")
16Ø  LET hj = CODE c$ (2) – 48 – 39 * (c$ (2) > "9")
17Ø  POKE a, 16 * hs + hj
18Ø  LET a = a + 1
19Ø  GO TO 12Ø
2ØØ  POKE a, 2Ø1
```

Das Ziel von Zeile 3Ø ist, daß wir einige Adressen vor dem Maschinencodeprogramm für die Aufnahme von Daten reservieren können; das ist oft nützlich. Zeilen 6Ø–8Ø füllen diesen freien Raum mit Nullen; die eigentlichen Daten können später, sobald sie gebraucht werden, mit POKE eingegeben werden. Zeilen 15Ø–16Ø wandeln den Hexcode so in zwei (dezimale) Zahlen zwischen Ø und 15 um (statt Ø – 9, A – F), daß 16 * hs + hj die tatsächliche Dezimalentsprechung des Hexcodes liefert. Zeile 17Ø setzt das mit POKE in die richtige Adresse.

Das Input-"s", kein Hexcode, dient als Begrenzer und teilt der Maschine mit, keinen Code mehr anzufordern. Zeile 2ØØ sorgt dafür, daß der letzte Befehl im Maschinencodeprogramm der RET- (spring zurück zu BASIC) Befehl ist, der Hexcode C9 hat (dezimal 2Ø1). Wie das Handbuch auf Seite 18Ø betont, ist es

wichtig, die Maschinencoderoutine mit diesem Befehl zu beenden; das Ladeprogramm setzt ihn deshalb automatisch, für den Fall, daß Sie es vergessen sollten. (Ein zusätzliches RET richtet keinen Schaden an; es macht also nichts, wenn es Ihnen doch einfällt und Sie ihn zweimal setzen.)

Maschinencode fahren

Um die Routine zu fahren, verwendet man den Befehl USR. Aus technischen Gründen ist USR eine *Funktion,* deren Argument die Adresse ist, wo die Maschinencoderoutine anfängt, und deren Wert der Inhalt des BC-Registers im Z8Ø, wenn die Routine aufhört. Unterwegs dorthin wird der Spectrum aber das Maschinencodeprogramm tatsächlich ausführen. (Das mag ein bißchen pervers klingen, aber es soll dazu führen, daß innerhalb eines BASIC-Programms Maschinencode leicht zugänglich wird.) Wenn der Maschinencode bei 32ØØØ beginnt, brauchen wir einen Befehl wie

```
LET y = USR 32ØØØ
```

obwohl wir uns selten für den tatsächlichen Wert von y am Ende des Ganzen interessieren! Jeder Befehl, der USR 32ØØØ verwendet und dessen Syntax richtig ist, geht. Beispiel:

```
RANDOMIZE USR 32ØØØ
```

(Beachten Sie, daß das Mühe spart, weil RANDOMIZE nur einen einzigen Tastendruck erfordert) oder auch

```
RESTORE USR 32ØØØ
```

oder was immer . . .

Wenn es Datenbytes gibt, muß 32ØØØ erhöht werden, um zu vermeiden, daß sie als Programmteil behandelt werden. Mit dem Folgenden, dem angefügt, was wir schon haben, können wir Maschinencode fahren. Es ist so eingerichtet, daß es in diesem Stadium auch andere Optionen zuläßt – siehe unten.

```
3ØØ  INPUT "Option?"; o$
31Ø  IF o$ = "r" THEN GO SUB 4ØØ
399  STOP
4ØØ  LET y = USR (b + d)
41Ø  RETURN
```

Maschinencode sichern

Ein Maschinencode-Programm läßt sich mit SAVE sichern, wenn man *Bytespeicherung* verwendet, siehe Handbuch Seite 144. Angenommen, unsere

Routine wird 77 Bytes lang, das letzte RET eingeschlossen. Wir sichern sie dann (etwa) unter dem Namen "m-code" mit dem Befehl

```
SAVE "m-code" CODE 32ØØØ, 77
```

wo 32ØØØ das Startbyte ist und 77 die Länge. Da wir die Positionierung ja normiert haben, ist es aber sinnlos, sich über die Länge den Kopf zu zerbrechen, und wir könnten regelmäßig 77 durch 6ØØ ersetzen. (Auf dem Magnetband dauert das ein bißchen länger, aber wirklich nicht viel.)
Eine solche Folge von gespeicherten Bytes wieder mit LOAD zurückzuladen, ist ebenso einfach. Der Befehl

```
LOAD "m-code" CODE
```

genügt. Wenn Sie den Namen vergessen haben, nehmen Sie

```
LOAD "" CODE
```

Wir können die beiden Dinge ebensogut als Optionen in unseren Lader einbauen:

```
32Ø  IF o$ = "s" THEN GO SUB 5ØØ
33Ø  IF o$ = "l" THEN GO SUB 6ØØ
5ØØ  INPUT "Name fuer SAVE?"; n$
51Ø  IF n$ = "" THEN LET n$ = "m-code"
52Ø  SAVE n$ CODE b, 6ØØ
53Ø  RETURN

6ØØ  INPUT "Name fuer LOAD?"; n$
61Ø  LOAD n$ CODE
62Ø  RETURN
```

Überprüfen und anzeigen

Ein letzter Zusatz ermöglicht uns, das Listing auf dem Schirm zu überprüfen oder sogar auszudrucken, wenn wir einen Drucker haben. (Wenn es Fehler gibt, korrigieren Sie die durch Eingabe mit POKE an diesen Adressen. HELPA, Anhang 7, macht das auf sinnvollere Weise, aber im Augenblick reicht das aus.)

```
34Ø  IF $ = "z" THEN GO SUB 7ØØ
35Ø  IF o$ = "p" THEN GO SUB 7ØØ
```

```
700  IF o$ = "p" THEN PRINT: IF o$ = "z" THEN LPRINT
702  FOR i = b TO a
710  LET j = PEEK i: LET js: INT (j/16): LET jj = j – 16 * js
720  LET h$ (1) = CHR$ (js + 48 + 7 * (js > 9))
730  LET h$ (2) = CHR$ (jj + 48 + 7 * (jj > 9))
740  IF o$ = "p" THEN PRINT i; "□□" + h$ + "□□"; j
750  IF o$ = "z" THEN LPRINT i; "□□" + h$ + "□□"; j
760  NEXT i
770  RETURN
```

Das liefert vom Programm sowohl ein Dezimal-, wie ein Hex-Listing der Adressen.

Beachten Sie, daß die Originaleingabe die Buchstaben a–f in *Kleinschreibweise* anzeigt, das Obige aber *Großschreibweise* verwendet. Wenn Sie das stört, ist es eine gute Übung, einen Weg zu finden, das zu verhindern. Aber in Wirklichkeit ist es sogar sehr praktisch, wechselweise Groß- und Kleinschreibung zu verwenden. Ich werde das auch weiterhin so halten.

Wir können diese Option nutzen, um das Listing vor und nach einem Lauf zu prüfen, vorausgesetzt wir fügen an

```
370  GO TO 300
```

damit wir die Optionen alle wieder zur Verfügung haben. Um anzuhalten, könnten Sie hinzufügen

```
360  IF o$ = "a" THEN STOP
```

Die möglichen Optionen sind nun:

a	STOP
l	LOAD
p	PRINT (auf dem Schirm anzeigen)
r	RUN (Maschinencode)
s	SAVE
z	COPY (an Drucker)

Der Rest des Buches geht davon aus, daß Sie das obige Programm auf Band gesichert haben und es für die gelieferten Maschinencode-Listings jederzeit verwenden können.

10 Arithmetik

Für den Anfang hier ein paar Routinen zu Addition und Subtraktion. Was könnte einfacher sein?

Der Schlüssel zur Arithmetik ist das Vorhandensein von ADD- und SUB-Befehlen, die beide das A-Register benützen und außer der direkten jede Adressierart verwenden können.

Schreiben wir zum Anfang ein Programm, das die Zahlen 4 und 7 zusammenzählen soll. Wir müssen die eine in das A-Register, die andere in das B-Register laden, sie addieren und das Resultat irgendwo hineinsetzen. Wenn wir den LADER mit 1 Datenbyte verwenden, können wir die Summe in die Adresse 32ØØØ (7DØØ) setzen. Zuerst übertragen in Opcode-Mnemonik:

4 ins A-Register setzen:	LD A, Ø4
7 ins B-Register setzen:	LD B, Ø7
Addieren (wobei die Summe	
im A-Register landet):	ADD A, B
Ergebnis speichern:	LD (7DØØ), A

Schlagen Sie die Opcodes nun im Anhang nach. Sie erhalten die Hexcodierung:

LD A, Ø4	3E Ø4
LD B, Ø7	Ø6, Ø7
ADD A, B	8Ø
LD (7DØØ), A	32 ØØ 7D

Beachten Sie, wie das 7DØØ die Reihenfolge ØØ7D annimmt, wie ich eben erwähnt habe.

Jetzt wird das geladen. Bringen Sie das LADER-Programm in den Spectrum und fahren Sie mit RUN. Wenn Datenbytes verlangt werden, geben Sie 1 ein. Dann der Reihe nach die Hexcodes eingeben, in der Form

```
3e  Ø4  Ø6  Ø7  8Ø  32  ØØ  7d
```

(weil der LADER in Kleinbuchstaben anzeigt) und schließen Sie ab mit

```
s
```

als Begrenzer (und um den entscheidenden RET-Befehl anzufügen, den ich vergessen habe!).

Als Option nehmen Sie "r" für Run.

Schön, aber wo ist die Lösung? Am leichtesten sieht man sie mit Option "p", einem Listing. Das Ergebnis ist dann:

32ØØØ	ØB	11
32ØØ1	3E	62
32ØØ2	Ø4	4
32ØØ3	Ø6	6
32ØØ4	Ø7	7
32ØØ5	8Ø	128
32ØØ6	32	5Ø
32ØØ7	ØØ	Ø
32ØØ8	7D	125
32ØØ9	C9	2Ø1

Wir sehen das Programm ab 32ØØ1, auch das abschließende RET, C9. Wir sehen ferner das Resultat der Addition, 11, in 32ØØØ, wo wir es haben wollten.

Bevor wir weitergehen, schreiben Sie Maschinencodeprogramme für die Berechnung von:

18 + 66

13 + 17

23 + 6

Sie verwenden das Obige, ändern Ø4 und Ø7 aber zu dem neuen Zahlenpaar ab; achten Sie darauf, daß in jedem Fall die Lösung in 32ØØØ steht.

Daten besser nutzen

Das ist gut, aber wir wollen nicht für jedes Zahlenpaar ein neues Programm schreiben, oder?

Wir könnten Ø4 und Ø7 zu allgemeinen Zahlen m und n abändern und schreiben

POKE 32ØØ2, m: POKE 32ØØ4, n

Wenn wir das machen, bringen wir aber Programm und Daten durcheinander, weil die zwei Zahlen eigentlich Daten sein müssen. Hier spielt das zwar keine Rolle, aber bei einem komplizierteren Programm ist es ärgerlich, wenn während eines Laufs das Programm selbst verändert wird (man spricht dann von einem *selbstmodifizierenden* Programm): Erstens, weil man es dann nicht mehr verwenden kann, zweitens, weil es keinen Nachweis dafür gibt, wie es vorher aussah.

Es wäre also klüger, die beiden Zahlen so festzulegen, daß sie als Daten addiert werden; man lädt sie als Teil des Programms in die Register, addiert sie und lädt die Lösung in Daten.

Wir reservieren deshalb drei Datenbytes:

32ØØØ	Erste Zahl
32ØØ1	Zweite Zahl
32ØØ2	Summe

was in Hex 7DØØ, 7DØ1 und 7DØ2 ist. Das Programm, das bei 7DØ3 beginnt sollte so aussehen:

Erste Zahl nach A laden:	LD A, (7DØØ)
Zweite Zahl nach B laden:	LD B, (7DØ1)
Addieren:	ADD A, B
Summe nach 32ØØØ stellen:	LD (7DØ2), A

Das ist wunderbar, aber wenn Sie sich die Hexcodes ansehen, entdecken Sie, daß es keinen Befehl LD B (nn) gibt. Mist!

Das kann man umgehen. Ein Weg ist der, über das HL-Register indirekt zu steuern, so daß wir statt LD B (7DØ1) schreiben:

HL mit der Adresse laden:	LD HL, 7DØ1
B durch HL laden:	LD B, (HL)

Das Programm hat also jetzt die Form:

LD A (7DØØ)	3AØØ7D
LD HL, 7DØ1	21Ø17D
LD B, (HL)	46
ADD A, B	8Ø
LD (7DØ2), A	32Ø27D

Laden Sie das, indem Sie drei Datenbytes reservieren, dann bringen Sie das Programm mit STOP zum Stehen, um die Daten zu setzen mit

POKE 32ØØØ, 44: POKE 32ØØ1, 33

wenn Sie (sagen wir) 44 und 33 addieren wollen. Dann GO TO 3ØØ, um in den LADER zurückzukommen, und die Option "r" für Run eingeben. Anschließend tippen Sie "p", um ein Listing zu erhalten. Die ersten drei Eintragungen lauten:

32ØØØ	2C	44
32ØØ1	21	33
32ØØ2	4D	77

Dann folgt der Rest des Programms. Die Summe 77 findet sich, wie wir gehofft hatten, in Adresse 32ØØ2.

Verändern Sie die mit POKE angegebenen Daten und sehen Sie sich die Ergebnisse an.

Versuchen Sie vor allem

```
POKE 32ØØØ, 24Ø: POKE 32ØØ1, 1ØØ
```

Sie werden sehen, daß der Computer überzeugt ist, 24Ø + 1ØØ = 84. Wenn Ihnen das merkwürdig vorkommt, haben Sie recht; aber das ist Ihre Schuld und liegt nicht am Computer! Der ADD-Befehl vergißt Übertragungsziffern. Stellen Sie sich das binär vor:

```
24Ø        1 1 1 1 Ø Ø Ø Ø
1ØØ      + Ø 1 1 Ø Ø 1 Ø Ø
           ---------------
      ┌─── Ø 1 Ø 1 Ø 1 Ø Ø = 84
      │    ---------------
      │    1 1
      ↓
      1
```

Die Summe erzeugt ein neuntes Bit, das ein 8 Bit-Byte nicht mehr aufnehmen kann; es fällt hinten herunter, und das mitgeteilte Ergebnis ist um den Wert des neunten Bits, nämlich 256, zu klein. In der Tat ist 256 + 84 = 34Ø, also die richtige Lösung. Keine Überprüfung ist erfolgt, keine hilfreiche Fehlermeldung angezeigt worden. Wenn Sie Maschinencode schreiben, sind Sie auf sich selbst gestellt. Was Sie nicht testen, erfahren Sie auch nicht.

Es gibt aber Methoden, mit Übertragsziffern und dieser Art von "Überlauf" fertigzuwerden, siehe weiter unten bei "Flaggen". Aber über dieses Problem brauchen wir uns jetzt noch nicht den Kopf zu zerbrechen.

Subtraktion

Jetzt wollen wir sehen, ob Sie das Programm so abändern können, daß es die zweite Zahl von der ersten *abzieht.* Sie brauchen aus dem

```
ADD A, B                  8Ø
```

nur ein

```
SUB A, B                  9Ø
```

zu machen und fortzufahren wie vorher. Probieren Sie das aus, vergewissern Sie sich, daß 44 – 33 = 11 und stellen Sie fest, welche Wirkungen Sie durch Überläufe erhalten (nehmen Sie etwa 17 – 99).

11 Ein Teilsatz von Z80-Befehlen

Es gibt 694 Z8∅-Befehle – hier eine Auswahl der wesentlichen und leichter zugänglichen, und was sie leisten.

Ich werde nicht jeden einzelnen der 694 Opcodes beschreiben, über die der Z8∅ verfügt; das wäre mühsam und unnötig. (Siehe dazu Anhang 4.) Wir sehen uns einen Teil von rund 3∅ Befehlsarten an (die ungefähr 23∅ konkrete Befehle umfassen). Leider können nicht alle sämtliche Adressierarten verwenden. Hier eine Tabelle zum schnellen Nachschlagen, die zeigt, welche Befehle was benützen können; die Opcodes stehen in Anhang 6.

Adressierart \ opcode	LD	ADD ADC SUB SBC AND OR XOR CP	INC DEC SLA SRA SRL	JR JRC JRNC JRZ JRNZ DJNZ	JP	JPZ JPNZ JPC JPNC JPP JPM	LD	ADD ADC SBC	INC DEC PUSH POP
Register	LD r, s	ADD A, r	INC r					ADD HL, r	INC r
Sofort	LD r, n	ADD A, n			JP nn	JPZ nn	LD r, nn		
Direkt	LDA, (nn) LD (nn), A						LD HL, (nn) LD (nn), HL		
Indirekt	LD A, (HL) LD (HL), A	ADD A, (HL)	INC (HL)		JP (HL)				
Indiziert	LD A, (IY + d) LD (IY + d), A	ADD A, (IY + d)	INC (IY + d)	JR d					

8 Bit-Operationen (Spalten LD bis JR) — 16 Bit-Operationen (Spalten JP bis INC/DEC/PUSH/POP)

Die Schreibweise in der Tabelle bedarf der Erklärung. Manche Opcodes mögen fremdartig erscheinen; mit denen befassen wir uns später. Im übrigen wird so vorgegangen:

1. Jede Eintragung in der Tabelle zeigt ein Beispiel für das Format der Befehlsart. Jeder andere Opcode in dieser Spalte kann ebenso verwendet werden.
2. "r" oder "s" meint jedes beliebige Register. Ob das ein 8 Bit- oder 16 Bit-Register ist, hängt davon ab, in welchem Teil der Tabelle der Befehl steht. Beispiel: Im Befehl LD r, s sind r und s jedes 8 Bit-Register (A, B, C, D, E, H oder L) aber in ADD HL, r ist r ein Registerpaar von BC, DE, HL, SP.
3. "n" ist jede 8 Bit-Zahl. "nn" ist jede 16 Bit-Zahl.
4. Wenn ein Register wie bei LD A, (nn) ausdrücklich angegeben ist, handelt es sich um das einzige Register, das für diesen Zweck verwendet werden kann.

Das ist arg vereinfacht. Manchmal sind andere Register verwendbar, aber der springende Punkt ist der, daß die Befehle, die ich gezeigt habe, stets in Ordnung sind und Sie sich Gedanken darüber machen können, Ihren Befehlswortschatz zu erweitern, wenn Sie diesen Teil hier gut beherrschen.

5 "d" ist jede 8 Bit-Zahl, aber sie wird stets zu einem 16 Bit-Wert hinzugefügt. Mit anderen Worten, es handelt sich um eine Indexverschiebung.

Sehen wir uns jetzt die neuen Opcodes an:

AND

Diese Operation nimmt den Inhalt des A-Registers und ein anderes 8 Bit-Feld und untersucht sie Bit für Bit. Nur wenn zusammengehörige Bits beide "1" sind, stellt er an diese Position im A-Register eine 1 zurück, im anderen Fall eine "Ø".

Beispielsweise hat AND A, Ø7 folgende Wirkung:

A-Register vor der Operation:	Ø Ø 1 1 Ø 1 Ø 1
Ø7:	Ø Ø Ø Ø Ø 1 1 1
A-Register nach dem AND:	Ø Ø Ø Ø Ø 1 Ø 1

Sehen Sie, wie die drei jüngeren Bits übertragen worden sind? Sie können AND also dazu verwenden, Teile eines Bytes auszuwählen.

OR

Das geht ähnlich wie AND, aber diesmal ist das Bit, das dabei herauskommt, eine "1" dann, wenn eines der anfänglichen Bits eine "1" ist. Ein OR A, Ø5 ergibt demnach:

A-Register vorher:	Ø 1 Ø Ø 1 Ø 1 1
Ø5:	Ø Ø Ø Ø Ø 1 Ø 1
A-Register nachher:	Ø 1 Ø Ø 1 1 1 1

Hier werden bestimmte Bits ohne Rücksicht auf ihren vorherigen Wert zu "1" gezwungen.

XOR

Hier müssen die ursprünglichen Bitwerte verschieden sein, damit das Ergebnis eine "1" wird. XOR A, B3 ergibt:

A-Register vorher:	Ø 1 Ø 1 1 Ø 1 Ø
B3:	1 Ø 1 1 Ø Ø 1 1
A-Register nachher:	1 1 1 Ø 1 Ø Ø 1

Das ist besonders nützlich, wenn man ein Register von Ø zu 1 und wieder zurück kippen möchte. Enthält das A-Register zu Anfang Ø, wird jedesmal bei Ausführung des Befehls XOR A, Ø1, der Wert im A-Register gekippt. (Ø zu 1, zurück zu Ø, wieder zurück zu 1 und so weiter.)

CP

Das ist der Befehl "Compare" = Vergleiche. Der Inhalt des A-Registers wird verglichen mit dem eines anderen 8 Bit-Feldes. Das wirft aber ein Problem auf: Wie wird das Ergebnis des Vergleichs angezeigt?

Dafür findet das F- (oder *Flaggen*) Register Verwendung. Jedes Bit des F-Registers enthält Information über die Wirkung des letzten Befehls, sie zu ändern. (Nicht alle Befehle verändern sie.)

Die Flaggen, die uns am meisten interessieren, sind die Übertrags-, Null-, Überlauf- und Vorzeichenflagge. CP kann jede davon verändern, aber die größte Bedeutung hier hat die Nullflagge, die dann gesetzt wird, wenn die beiden verglichenen Werte gleich sind.

Ist der Inhalt des A-Registers *kleiner* als der des damit verglichenen Bytes, wird die Vorzeichenflagge gesetzt. Das heißt soviel wie "das Ergebnis ist negativ". Mehr brauchen Sie über die Flaggen im Augenblick nicht zu wissen; das ist ein verwickeltes Thema, wenn man sich näher darauf einläßt. Siehe Kapitel 16 und Anhang 5 für zusätzliche Informationen.

Die Sprünge

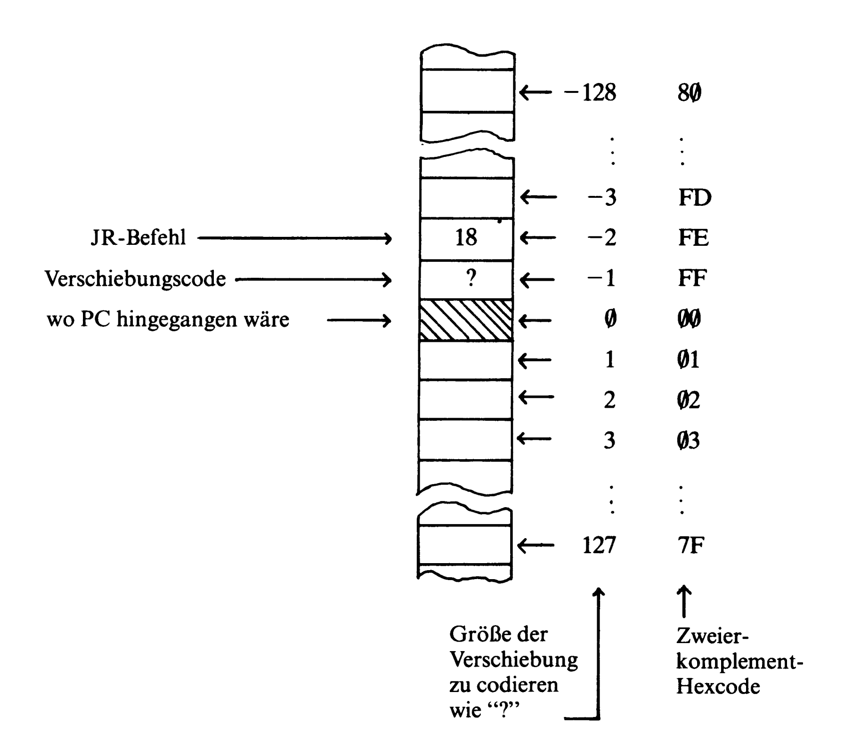
Alle bedingten Sprünge verzweigen (oder tun es nicht) entsprechend dem Inhalt der Flaggen. So bedeutet etwa JPZ "spring, wenn die Nullflagge gesetzt ist.". Jetzt können wir erkennen, wie der CP-Befehl verwendet werden kann. Nehmen wir etwa an, wir möchten sehen, ob ein bestimmtes Byte, auf das HL zeigt, 1E hex enthält. Wenn das der Fall ist, wollen wir verzweigen zu 447B. Der Code lautet:

```
LD A, 1E        3E1E
CP A, (HL)      BE
JPZ 447B        CA7B44
```

Alle anderen Sprünge verhalten sich ähnlich; JPNZ heißt "spring zu einem Ergebnis, das nicht Null ist" (Nullflagge *nicht* gesetzt), JPP "spring zu einem positiven Ergebnis" (Vorzeichenflagge *nicht* gesetzt), JPM "spring zu einem negativen Ergebnis" (Vorzeichenflagge gesetzt), JPNC "spring zu keinem Übertrag" (Übertragsflagge *nicht* gesetzt) und so weiter. Alle haben eines gemeinsam, nämlich, daß die Sprungadresse feststeht. Mit anderen Worten: Wenn wir aus irgendeinem Grund eine Routine irgendwo anders im Speicher fahren wollen als dort, wo wir sie zunächst geladen haben, müssen alle Sprungadressen geändert werden. Der Z8Ø bewältigt das geschickt, indem er "relative Sprünge" (JR) zuläßt. Anders ausgedrückt: Sie können von dort aus, wo Sie sich befinden, soundsoviele Bytes vorwärts (oder rückwärts) springen. Diese Verschiebung ist (in Zweierkomplementation, Anhang 1) in 1 Byte enthalten, so

daß die Entfernung, die übersprungen werden kann, rückwärts 128 oder vorwärts 127 Bytes nicht zu übersteigen vermag.

Die Verschiebung wird danach berechnet, wohin der PC-Wert als nächstes gegangen *wäre,* wenn kein Sprung stattgefunden hätte, nämlich von der Adresse des nächsten Befehls im Programm aus. Etwa so:



Hier ein Beispiel. Wir wollen jedes Speicherbyte der Reihe nach untersuchen, bis das erstemal 1E hex auftaucht. Nehmen wir der Einfachheit halber an, die Startadresse sei schon in HL. Wir könnten schreiben:

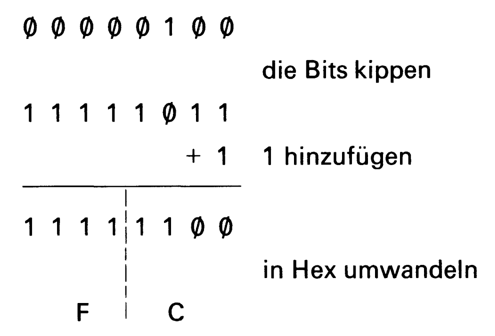
```
          LD A, 1E
Schleife: CP A, (HL)
          INC HL
          JRNZ Schleife
```

Zwei Punkte sind zu erklären. Erstens habe ich einen neuen Befehl eingeschmuggelt: INC. Das ist eine Abkürzung für INCrement = Inkrementieren (um 1 erhöhen). Es fügt dem Inhalt des angegebenen Registers einfach 1 hinzu, so daß die Vergleichsoperation stets das nächste Speicherbyte betrachtet, weil HL mit jeder Schleife um 1 angehoben wird. (Übrigens leistet DEC für DECrement = Dekrementieren genau das Gegenteil.) Der zweite Punkt: Es gibt keinen

offenkundigen Unterschied zwischen JRNZ *Schleife* und JPNZ *Schleife.* Erst wenn wir die Befehle in Maschinencode *assemblieren,* wird der Unterschied deutlich. Nehmen wir an, der Code wird, wie gewohnt, von 7DØØH aus geladen:

Adresse		Befehl	Hexcode
7DØØ		LD A,1E	3E1E
7DØ2	Schleife:	CP A,(HL)	BE
7DØ3		INC HL	23
7DØ4		JRNZ Schleife	2ØFC

Warum steht im Adreßteil des JRNZ-Befehls FC? Das geht so: Bei Ausführung des JRNZ-Befehls wird der PC um 2 erhöht, weil es sich um einen 2 Byte-Befehl handelt. Der PC steht jetzt also bei 7DØ6. Wir wollen zu *Schleife* springen, die bei 7DØ2 ist, 4 Bytes dahinter oder −4 Bytes entfernt, um die Denkweise des Z8Ø zu übernehmen. Nun ist 4 in Binär ØØØØØ1ØØ, und wir erzeugen −4 durch Kippen der Bits und Hinzufügen von 1 (Zweierkomplement, ja?). Demnach:



Noch etwas, das Ihnen vielleicht Kopfzerbrechen bereitet: INC verändert die Flaggen nicht, so daß ich nach dem Inkrementieren unbesorgt testen kann.

Dasselbe Programm mit absoluten Sprüngen hätte so ausgesehen:

Adresse		Befehl	Hexcode
7DØØ		LD A,1E	3E1E
7DØ2	Schleife:	CP A,(HL)	BE
7DØ3		INC HL	23
7DØ4		JPNZ Schleife	C2Ø27D

Beachten Sie, daß der JPNZ-Befehl 3 Bytes umfaßt, weil er eine ganze 16 Bit-Adresse enthält; und vergessen Sie nicht, die 2 Bytes dieser Adresse zu vertauschen!

In der Sprunggruppe, die ich noch nicht erwähnt habe, gibt es noch einen sehr wirkungsvollen Befehl – DJNZ. Er dekrementiert das B-Register um 1 und springt (relativ) nur, wenn das Ergebnis nicht Null ist.

Nehmen wir an, unser kleines Programm "suche 1E" solle nur einen Bereich von einhundert Bytes (Hex 64) Länge absuchen und danach die Schleife verlassen, ob es 1E gefunden hat oder nicht:

```
            LD B, 64        Ø6, 64
            LD A, 1E        3E1E
Schleife:   CP A, (HL)      BE
            JPZ hab dich    CA – – (Adresse für hab dich)
            INC HL          23
            DJNZ Schleife   1ØF9
```

Die Schleife wird hundertmal ausgeführt, es sei denn, ein 1E wird gefunden, worauf eine Verzweigung zu *Hab dich* erfolgt. Anders ausgedrückt: DJNZ wirkt wie eine schlichte FOR-Schleife in BASIC.

Beachten Sie, daß bei allen relativen Sprungbefehlen JR, JRC, JRNC, JRNZ und JRZ die Sprunggröße auf dieselbe Weise berechnet wird. Eine Tabelle von Zweierkomplement-Hexcodes finden Sie in Anhang 1 für das Codieren von Sprüngen per Hand; unser Nutzprogramm HELPA in Anhang 7 errechnet relative Sprünge automatisch für Sie, während der Code geschrieben wird.

ADC und SBC

Das sind die Befehle "ADD with Carry" = Addiere mit Übertrag und "SUB with Carry" – Subtrahiere mit Übertrag. Ich habe vorhin darauf hingewiesen, daß das Flaggenregister eine Übertragsflagge enthält. Sie wird gesetzt, sobald durch einen arithmetischen Befehl aus einem Register ein Übertrag erzeugt wird. Der ADC-Befehl wirkt genau wie ADD, fügt aber 1 hinzu, falls das Übertragsbit durch eine vorherige Operation gesetzt worden ist. Der SBC-Befehl wirkt genauso, nur zieht er die Übertragsflagge ab.

Die Verschiebungen

Die Verschiebebefehle SLA, SRA, und SRL haben alle die Wirkung, Bitmuster umherzuschieben.

SLA verschiebt das Muster um 1 Bit nach links. Enthält das B-Register also

Ø	Ø	1	Ø	1	1	Ø	Ø

und es wird SLA B ausgeführt, führt das zu

Ø	1	Ø	1	1	Ø	Ø	Ø

(Beachten Sie, daß rechts zum Auffüllen eine Null verwendet wird.)

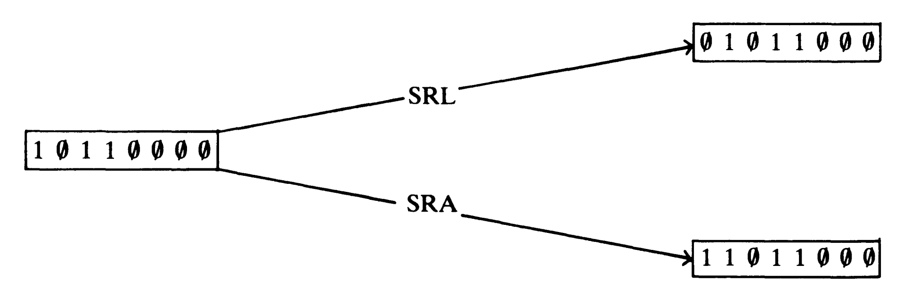
Da ØØ1Ø11ØØ = 44 und Ø1Ø11ØØØ = 88 (dezimal), ist die Wirkung erkennbar die, daß mit 2 multipliziert wird.

Ein weiteres SLA B ergibt:

1 Ø 1 1 Ø Ø Ø Ø

Da das ältere Bit jetzt 1 ist, wird das als negative Zahl betrachtet und die Vorzeichenflagge gesetzt. Für den Programmierer: Der Wert (176) kann nicht von einem Byte aufgenommen werden, so daß ein Überlauf vorliegt.

Die Rechtsverschiebung geht ganz ähnlich, nur muß man sich eines merken: SRL füllt das ältere Bit mit einer Null auf, SRA aber mit dem, was vorher schon da war. Ein Beispiel dafür sehen Sie anschließend:

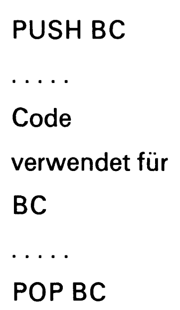


Der Grund dafür: SRL ist eine *logische Rechtsverschiebung,* die das Bitmuster nur verschiebt, ohne es zu ändern. SRA dagegen ist eine *arithmetische Rechtsverschiebung;* die Operation wird behandelt als "teile durch 2". Wenn eine negative Zahl durch 2 geteilt wird, muß auch das Resultat negativ sein, so daß wir das Vorzeichenbit beibehalten müssen.

PUSH und POP

Sie werden sich an diese Begriffe wohl von unserer Besprechung der Stapel her erinnern. Hier werden sie genau auf dieselbe Weise verwendet und erlauben uns Zugang zum Maschinenstapel auf andere Weise als durch den Aufruf einer Subroutine.

Das kann nützlich dafür sein, Werte zeitweilig zu sichern. Angenommen, Sie haben einen Wert in BC, den Sie später brauchen, wobei Sie jetzt aber BC für etwas anderes nutzen wollen. Sie können schreiben:



Das wird oft auch vor einer Subroutine CALL gemacht, also spielt es keine Rolle, welche Register das Unterprogramm verwendet; die Daten des Rufprogramms können dadurch nicht beeinflußt werden. Der Code könnte dann etwa so aussehen:



Voraussetzung ist, daß das A-Register von der Routine manipuliert wird, so daß wir es nicht sichern müssen.

Vorsicht

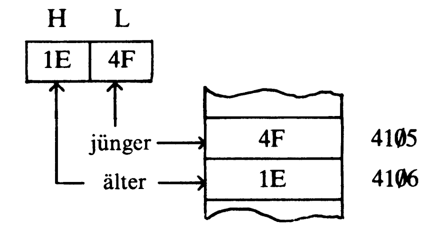
Wenn Sie nicht bewußt etwas ändern wollen, wird der Stapelzeiger SP entsprechend dem Betriebssystem des Spectrum gesetzt. Es schadet nicht, ihn bei diesem Wert zu belassen, vorausgesetzt, Sie sorgen dafür, daß PUSH- und POP-Anweisungen paarweise ausgeglichen sind, so daß SP beim Verlassen der Maschinencode-Routine zu seinem Ursprungswert zurückkehrt. Ebenso müssen CALL- und RET-Befehle einander entsprechen. (USR erzeugt ein CALL, ausgeglichen durch das abschließende RET, das vom LADER-Programm am Ende angefügt wird.)

Eine 16 Bit-Eigenheit

Ein besonders wichtiges Merkmal der 16 Bit-Operation (vor allem PUSH, POP, LD) ist die Reihenfolge, in der Bytes vom Register in den Speicher und umgekehrt übertragen werden. Das ist so:

LD (41Ø5), HL

hat die folgende Wirkung, wenn HL 1E4F enthält:



Anders ausgedrückt: Das weniger bedeutsame oder "jüngere" Byte im Register wird in die angegebene Adresse geladen, und das bedeutsamste oder "ältere" Byte in das nachfolgende. Umgekehrt hätte

LD HL, (41Ø5)

genau die entgegengesetzte Wirkung. (NB: Der Code ist nach der üblichen Verfahrensweise 2AØ541!). Ebenso hat

LD HL, 1ØØØ

(ein Versuch, HL mit dem Wert 1ØØØ hex zu laden) den Code

21ØØ1Ø

so daß, obwohl es sich bei 1ØØØ um Daten und nicht um eine Adresse handelt, die Bytes wie gewohnt übertragen werden.

Abstürze

Wenn ein BASIC-Programm abstürzt, können Sie mit BREAK so oder so immer heraus, ohne das Programm zu verlieren. Maschinencode-Abstürze sind schlimmer. Entweder löschen sie im Speicher alles (wie NEW) oder das Gerät bleibt völlig stecken und muß dadurch neu gesetzt werden, daß man kurz den Stromstecker herauszieht – mit demselben Ergebnis. Wollen Sie einen Absturz sehen? Hier ist der einfachste, den ich kenne und der nicht NEW bewirkt:

RANDOMIZE USR 996

Geräusche, Farben und keine Reaktion vom Keyboard. Puh. RANDOMIZE USR 14ØØ ist noch schlimmer.

(Der alte Dampf-ZX81 lieferte großartige Op-Art-Abstürze; der Spectrum ist besser geschützt und bleibt in der Regel auf recht zivilisierte Weise hängen oder schnappt sich den Ball und will überhaupt nicht mehr mitspielen.)

Bei der Arbeit mit Maschinencode sind Abstürze unvermeidlich, das werden Sie bald merken. Das einzig sichere Heilmittel ist, den Stromstecker zu ziehen, worauf Sie von vorn anfangen müssen. Es gibt aber ein paar lohnende Vorsichtsmaßnahmen zur Verringerung der Gefahr:

1 Prüfen Sie alle Maschinencode-Listings pedantisch genau und vergewissern Sie sich, daß Sie sie richtig eingegeben haben.
2 Verwenden Sie *nicht* HALT (Opcode 76 hex). Das ist *nicht* dasselbe wie der BASIC-Befehl STOP – es kommt nur zu einem Absturz.
3 Achten Sie darauf, daß CALL- und RET-Befehle einander ebenso entsprechen wie PUSH- und POP-Anweisungen.
4 Rufen Sie die *richtige* Startadresse auf.
5 Sichern Sie mit SAVE auf Band, was Sie können, bevor Sie die Routine zum erstenmal ausprobieren, es sei denn, daß nicht viel verlorengehen kann.

12 Ein Maschinencode-Multiplikator

Maschinencode enthält keinen Befehl für das Multiplizieren von Zahlen, aber es geht trotzdem, wenn Sie Arithmetik, Logik und Verschiebungen verbinden. Wir gehen den Dingen schon ein bißchen mehr auf den Grund . . .

Schreiben wir ein paar einfache Routinen. Sie erinnern sich an meine Bemerkung, daß es für den Z8Ø keinen Multiplikationsbefehl gibt? Verfassen wir eine Subroutine, die das bewältigt.

Ein Beispiel

Als erstes sollten wir uns die Art des Problems ansehen. Das geht am besten mit einem Beispiel. Wir machen es so einfach wie möglich und arbeiten mit 8 Bit-Registern. Wenn wir also 9 mit 13 multiplizieren wollen, sieht das so aus:

```
  ØØØØ1ØØ1
x ØØØØ11Ø1
```

Das können wir nun als eine übliche lange Multiplikation behandeln, aber weil sie in Binär ist, geht das sogar einfacher als sonst: Ist die laufende Ziffer, die wir malnehmen, 1, wird die obere Zeile abgeschrieben, ist sie Null, tun wir gar nichts:

```
  ØØØØ1ØØ1     P
x ØØØØ11Ø1     Q
  --------
  ØØØØ1ØØ1
  ØØ1ØØ1ØØ
  Ø1ØØ1ØØØ
  --------
  Ø111Ø1Ø1
```

Natürlich mußten wir bei jedem Schritt rechts Nullen anfügen, wie bei einer langen Dezimalmultiplikation auch. Nach den Begriffen des Maschinencodes entspricht das einer Linksverschiebung. Ich habe den beiden Zahlen die Namen P und Q gegeben.

Während P nach links verschoben wird, empfiehlt es sich, Q gleichzeitig nach rechts zu verschieben, weil wir auf diese Weise nur das jüngere Bit von Q untersuchen müssen, um zu erfahren, ob P zur Summe addiert werden muß oder nicht.

Prozedur

Nehmen wir an, P und Q stehen in den Registern D und E. Die Prozedur geht dann so:

1	A-Register auf Null setzen.	
2	Ist bei E das jüngere Bit 1, D nach A addieren.	wiederhole diese Schritte achtmal
3	D nach links verschieben	
4	E nach rechts verschieben.	

Hier ein erster Versuch, das zu codieren:

```
LD A, ØØ
LD B, Ø8
```

Der erste Schritt ist naheliegend; der zweite setzt B als Schleifenzähler in Verbindung mit einem DJNZ, das am Ende erscheinen soll. Nun wollen wir das jüngere Bit von E prüfen. Das können wir zur Zeit nur, wenn wir ein Maskenmuster (ØØØØØØØ1) mit einer AND-Operation verwenden. Setzen wir das C-Register auf dieses Muster:

```
LD C,Ø1          [Hexcodierung siehe unten]
```

Wir können eine AND-Verbindung nur mit dem A-Register herstellen, und dabei wird dessen Inhalt gelöscht, also sichern wir ihn zuerst in L:

```
Schleife:   LD L, A
```

nehmen dann das jüngere Bit von E heraus und stellen das A-Register wieder her:

```
LD A, C
AND A, E
LD A, L
```

War das Resultat von A Null, müssen wir um den Teil "addiere D nach A" von Schritt 2 so herumspringen:

```
JRZ Verschiebung
```

(Zur Beachtung: Da LD die Flaggen nicht beeinflußt, bezieht sich das JRZ weiter auf das AND.) Ansonsten führen Sie das ADD aus:

```
ADD A, D
```

Jetzt die Verschiebungen:

```
Shiftung:  SLA, D
           SRA, E
```

und stellen Sie fest, ob wir die Schleife schon oft genug ausgeführt haben:

```
           DJNZ Schleife
           (RET)
```

Der Code

Hier das ganze Ding:

```
           LD A,ØØ          3EØØ
           LD B,Ø8          Ø6Ø8
           LD C,Ø1          ØEØ1
Schleife:  LD L,A           6F
           LD A,C           79
           AND A,E          A3
           LD A,L           7D
           JRZ Shiftung     28Ø1
           ADD A,D          82
Shiftung:  SLA D            CB22
           SRA E            CB2B
           DJNZ Schleife    1ØF3
```

Wenn Sie dieses Programm ausprobieren wollen, müssen Sie dafür sorgen, daß die Register D und E die Werte enthalten, die multipliziert werden sollen. Sie könnten dem Programm also etwa voransetzen:

```
           LD HL,7DØØ       21ØØ7D
           LD D,(HL)        56
           INC HL           23
           LD E,(HL)        5E
```

und dann 7DØØ (hex) und 7DØ1 (hex) mit den zu multiplizierenden Werten über POKE eingeben, bevor das Programm aufgerufen wird. Diese beiden Bytes sind natürlich die beiden Nullbytes am Anfang der Routine, so daß das LD, HL, 7DØØ in 7DØ2 beginnt. Beachten Sie, daß ich dem Programm keine konkreten Adressen zugeteilt habe. Das kommt daher, daß alle Sprünge relativ sind, es auf konkrete Adressen also nicht ankommt, sondern nur auf Distanzadressen.

Sie müssen die Lösung auch *ausgeben;* im Augenblick sitzt sie im A-Register. Ein Weg ist der, für die Lösung noch ein Datenbyte 7DØ2 zu reservieren, wie ich das vorher beim Additionsprogramm gemacht habe. Das heißt, Sie müssen anfügen:

```
LD (7DØ2), A     32Ø27D
```

und beim Laden des Codes 3 Datenbytes verwenden. Um die Lösung zu erhalten, nehmen Sie

```
PRINT PEEK 32ØØ2
```

Sie können auch das Ergebnis vom A-Register ins C-Register übertragen durch

```
LD B, Ø          Ø6ØØ
LD C, A          4F
```

und den Maschinencode aufrufen mit

```
PRINT USR 32ØØ2
```

Denken Sie daran, daß USR eine Funktion ist und bei der Rückkehr zu BASIC den Wert des Registerpaars BC enthält; dieser Befehl fährt also zugleich den Maschinencode und zeigt die Lösung an! (Ich bin davon ausgegangen, daß wir wieder bei nur zwei Datenbytes sind; deshalb beginnen wir bei 32ØØ2.)

BIT

Ich muß ein Geständnis ablegen: Es gibt einen einfacheren Weg, zu prüfen, ob das jüngere Bit von E 1 enthält. Das leistet ein Befehl BIT Ø, E. So wird aus:

```
Schleife:  LD L, A       6F
           LD A, C       79
           AND A, E      A3
```

einfach:

```
Schleife:  BIT Ø, E      CB43
```

und das LD A, L muß ebenfalls verschwinden.

Warum ich Ihnen das nicht gleich gesagt habe? Tja, eigentlich hatte ich versprochen, nur den Teilbestand von Befehlen in der Tabelle zu verwenden, und daran habe ich mich nicht gehalten. Aber dabei habe ich einen wichtigen Punkt deutlich machen können: Man kann die Dinge befriedigend bewältigen, ohne den ganzen Befehlsvorrat zu kennen.

Das war eher ein theoretisches Beispiel; ich habe es gewählt, weil es mehrere gängige Befehle auf eine konventionelle, aber nicht unbedingt naheliegende Weise verwendet, will damit aber nicht sagen, daß Sie Bedarf an Dutzenden von Multiplikationen ganzer 8 Bit-Zahlen haben.

13 Das Bildschirm-Display

Ein großer Teil vom Spectrum-Speicher ist einer einzigen Aufgabe gewidmet: das Display auf dem Bildschirm aufrechtzuerhalten. Um das Display in einem Maschinencodeprogramm nutzen zu können, muß man begriffen haben, wie die Information angeordnet ist.

Die Information, die das Bildschirm-Display steuert, ist in zwei RAM-Bereichen enthalten, die *Displayfile* und *Attributfile* heißen. Das erste steuert Zeichen und hochauflösende Grafik, das zweite Farben und Dinge wie FLASH und BRIGHT. Sie sind auf recht unterschiedliche Weise aufgebaut. Die einfachere der beiden ist das Attributfile, mit dem ich anfange.

Attributfile

Es besetzt 24 * 32 = 768 Bytes an den folgenden Plätzen:

	Dezimal	Hex
Beginn des Attributfiles	22528	58ØØ
Ende des Attributfiles	23925	5AFF

Die ersten 22 * 32 = 7Ø4 bewältigen den üblichen PRINT-Bereich des Bildschirms, Reihen Ø–21 für PRINT AT-Befehle. Die letzten 2 * 32 = 64 sind für den unteren Bereich von zwei Zeilen zuständig, der normalerweise für Fehlermeldungen und Edit verwendet wird. Der PRINT-Bereich endet bei Adresse 23231 dezimal, 5ABF hex; der untere Bildschirmabschnitt beginnt bei 23232 (5ACØ).

Wenn die Reihen wie gewohnt von Ø bis 21 numeriert werden (wobei wir uns die unteren Abschnitte als zusätzliche Zeilen 22 und 23 vorstellen) und die Spalten von Ø bis 31, entspricht Reihe r, Spalte c

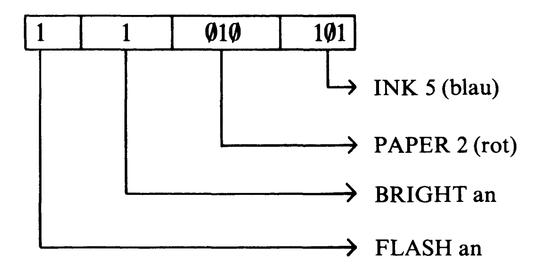
22528 + 32 * r + c

und diese Adresse enthält das *Attribut* für diese Position.

Jedes Attribut ist ein Einzelbyte, dessen acht Bits wie folgt aufgeteilt sind:

<table>
<tr><td>FLASH
ein/aus</td><td>BRIGHT
ein/aus</td><td colspan="3">drei Bits für PAPER farbe</td><td colspan="3">drei Bits für INK farbe</td></tr>
</table>

bei Ø für "aus" und 1 für "ein". Das Byte 11Ø1Ø1Ø1 teilt sich also folgendermaßen auf:



(was natürlich gräßlich aussieht). Wenn Sie dieses Attribut für die Position in Reihe 5, Spalte 7 wünschen sollten, müßten Sie das obige Byte speichern an

22528 + 32 * 5 + 7

also

22695.

Das Attribut selbst ist 213 dezimal; demnach speichert der Befehl

```
POKE 22695, 213
```

das Byte am richtigen Platz. Die Inkfarbe sehen Sie natürlich nicht, wenn nur ein Leerraum angezeigt werden soll, also müßten Sie etwa eingeben

```
PRINT AT 5, 7; "*"
```

damit das auch funktioniert.

Da jede Reihe 32 Spalten hat, findet man die Adresse für den Platz unmittelbar unter einem beliebigen anderen durch Addieren von 32 dezimal, 2Ø hex. Die Attribute sind also so angelegt:

oberste Reihe	58ØØ	58Ø1	58Ø2	...	581D	581E	581F	PRINT-Bereich
erste Reihe	582Ø	5821	5822	...	583D	583E	583F	
zweite Reihe	584Ø	5841	5842	...	585D	585E	585F	
.	.	.	.	...	.	.	.	
.	.	.	.	...	.	.	.	
.	.	.	.	...	.	.	.	
21. Reihe	5AAØ	5AA1	5AA2	...	5ABD	5ABE	5ABF	
22. Reihe	5ACØ	5AC1	5AC2	...	5ADD	5ADE	5ADF	Meldungen
23. Reihe	5AEØ	5AE1	5AE2	...	5AFD	5AFE	5AFF	

Displayfile

Es ist viel komplizierter, weil jedes Zeichen im Display als eine Liste von *acht* Bytes gespeichert wird, jede entsprechend einer Reihe der 8 x 8-Anordnung von hochauflösenden Pixels, die das Zeichen besetzt. Zu allem Übel sind diese *nicht* in der naheliegenden Reihenfolge gespeichert.

Wenn Sie mit LOAD SCREEN$ ein Bild in den Spectrum laden, müssen Sie die seltsame Reihenfolge bemerkt haben, in der das Bild "gemalt" wird. Tatsächlich ist das genau die Reihenfolge der Bytes im Displayfile. Am besten sieht man das, wenn man den Bildschirm mit schwarzer INK füllt, mit SAVE sichert und mit LOAD wieder lädt:

```
1Ø  FOR r = Ø TO 21
2Ø  PRINT "[32 × ■]"
3Ø  NEXT r
4Ø  SAVE "leer" SCREEN$
```

Nehmen Sie das auf Band, geben Sie ein

```
CLS: LOAD "leer" SCREEN$
```

und sehen Sie sich an, wie der Bildschirm geschwärzt wird.

Nun wollen wir stärker auf die Zahlen eingehen. Die Adressen sind:

	Dezimal	Hex
Beginn des Displayfiles	16384	4ØØØ
Ende des Displayfiles	22527	57FF

und die Gesamtzahl der Bytes beträgt 32 * 24 * 8 = 6144 dezimal, 18ØØ hex.

Am besten stellt man sich das Displayfile so vor, daß der Bildschirm in drei Blöcke aufgeteilt wird:

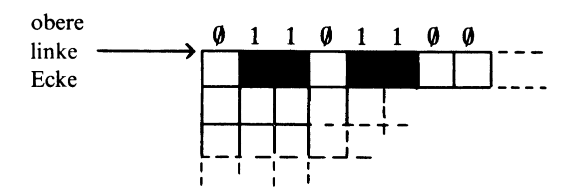
BLOCK 1	Reihen Ø–7 des Bildschirms
BLOCK 2	Reihen 8–15 des Bildschirms
BLOCK 3	Reihen 16–23 des Bildschirms

Beachten Sie, daß Block 3 die "Meldungs"-Reihen 22 und 23 einschließt. Jeder Block erfordert 6144/3 = 2Ø48 Bytes (8ØØ hex). Die ersten 2Ø48 des Displayfiles bewältigen Block 1, die nächsten Block 2, die letzten Block 3, und in jedem Block ist die Anordnung dieselbe. In Hex haben wir

Block	Startadresse	Endadresse
1	4ØØØ	47FF
2	48ØØ	4FFF
3	5ØØØ	57FF

Zum Verständnis der Anordnung innerhalb eines Blocks befasse ich mich mit Block 1; die anderen sind ähnlich (vergessen Sie aber nicht, daß Block 3 den "Meldungs"-Bereich einschließt). Der Prozeß läßt sich leichter beschreiben auf dem Hochauflösungs-Koordinatengitter mit 256 x 176 Pixels. Die Reihen werden von Ø bis 175 numeriert, die Spalten von Ø bis 255, mit Reihe 175 oben und Spalte Ø links, wie gewohnt für PLOT-Befehle *Spalte, Reihe.* Block 1 besteht dann aus den Reihen 175–112, insgesamt 8 x 8 = 64 Reihen zu je 256 Pixel.

Die ersten 32 Bytes des Displayfiles enthalten die Information für Reihe 175. Jedes Byte bestimmt einen Abschnitt von 8 Spalten. Das erste Byte betrifft die Spalten Ø–7 und verwendet je Pixel ein Bit. Enthält beispielsweise Adresse 4ØØØ das Byte Ø11Ø11ØØ, dann sind die ersten acht Pixel oben links am Bildschirm



mit Leerräumen für "Ø" und schwarzen Quadraten für "1".

Das nächste Byte bewältigt die Spalten 8–15, das danach 16–23 und so weiter, bis nach 32 Bytes die ganze Reihe 175 erfaßt ist.

Um das beobachten zu können, geben Sie CLS und

```
POKE 16384, BIN Ø11Ø11ØØ
```

und Sie sehen in der obersten Reihe zwei kurze Striche (die 11 in Binär.) Nehmen Sie

```
POKE 16384, BIN 1Ø1Ø1Ø1Ø
```

sehen Sie vier Punkte. Verändern Sie dann 163854 zu 16385, 16386 ... bis hin zu 16415 und füllen Sie Reihe 175 aus.

Das nächste Byte befindet sich in Adresse 16416. Wenn Sie

```
POKE 16416, BIN 1Ø1Ø1Ø1Ø
```

probieren, werden Sie sehen, daß das *nicht* die ersten acht Spalten der nächsten Reihe, also 174, sind! Es sind vielmehr die ersten acht Spalten von Reihe 167.

175 – 8 = 167 – also werden *acht* Hochauflösungs-Reihen übersprungen. Dieses Byte und die 31 danach füllen den Rest von Reihe 167 aus. Dann geht die Maschine weiter zu Reihe 167 – 8 = 159 und setzt seine Sprünge über 8 Reihen fort, bis sie an der Unterseite von Block 1 angekommen ist. Die Reihen haben also in Blöcken von 32 Bytes die Reihenfolge

175 167 159 151 143 135 127 119

Würde sie wieder acht überspringen, wären wir bei Reihe 111, die nicht mehr in Block 1 ist. (Es ist die oberste Reihe von Block 2.) Stattdessen springt der Spectrum zu den Reihen unmittelbar unter den eben behandelten hinauf, und die nächsten acht 32 Byte-Abschnitte befassen sich mit den Reihen

174 166 158 150 142 134 126 118

Dann mit den Reihen unter diesen:

173 165 157 149 141 133 125 117

und so weiter, bis schließlich die unterste Blockreihe erreicht ist:

168 160 152 144 136 138 12Ø 112

Damit ist Block 1 abgeschlossen. Die nächsten 2Ø48 Bytes verfahren mit Block 2 ähnlich, und zuletzt kommt Block 3 an die Reihe. Die Reihenfolge in jedem Block ist genau die gleiche: die Adressen verschieben sich um 2Ø48 nach oben, die Positionen um 64 Reihen auf dem Schirm nach unten, von einem Block zum nächsten.

Das sieht reichlich kompliziert aus. Am besten kommt man damit zurecht, wenn man sich zuerst Block für Block, dann Abschnitte von 8 Reihen und erst dann eine bestimmte Reihe vorstellt. Die Anordnung ergibt mehr Sinn, wenn wir uns daran erinnern, daß ein Zeichen ein Quadrat von 8 x 8 Pixeln (bei hoher Auflösung) ist; das heißt, eine PRINT AT-Position. Die acht Reihen des Zeichens werden dargestellt durch die zugehörigen acht Bytes (genau wie bei der benutzergewählten Grafik, siehe "Sinclair ZX Spectrum, Programmieren leicht gemacht" *, S. 67). Ein Zeichendisplay in Block 1 wird also in der Displaydatei folgendermaßen angeordnet:

32 Bytes für die oberste Reihe jedes Zeichens in Zeile Ø

32 Bytes für die oberste Reihe jedes Zeichens in Zeile 1

32 Bytes für die oberste Reihe jedes Zeichens in Zeile 7

und dann

32 Bytes für die *zweite* Reihe jedes Zeichens in Zeile Ø

32 Bytes für die *zweite* Reihe jedes Zeichens in Zeile 1

* Von Ian Stewart und Robin Jones, Birkhäuser, Basel.

32 Bytes für die *zweite* Reihe jedes Zeichens in Zeile 7

Dann weiter zur dritten, vierten, . . . , achten Reihe. Sieht zwar immer noch verwickelt aus, aber vielleicht nicht mehr ganz so schlimm wie auf den ersten Blick.

Ich gewöhne uns im nächsten Kapitel zuerst an die Attributdatei, und im Kapitel danach nehmen wir uns das Displayfile vor.

14 Das Attributfile

Den Bildschirm auf einen Schlag mit einer Farbe ausfüllen (oder mit mehreren); alle blinkenden Pixels abschalten; eine einzelne farbige Spalte abrollen . . .

Am einfachsten beginnt der Umgang mit dem Attributfile damit, daß man eine Maschinencode-Routine schreibt, um ein Quadrat einer bestimmten Farbe an der oberen linken Ecke anzuzeigen (wo das File beginnt, Adresse 5800 hex). Ich verwende ein Datenbyte (wie gewohnt 7D00) für die Farbe (Attributbyte), dann lautet der Code:

```
LD A, Attribut      3A007D
LD (5800), A        320058
```

Laden Sie das, geben Sie STOP und setzen Sie die Daten mit

POKE 32000, 32

(Da 32 = 4 * 8 ist das Attribut PAPER 4.) nun GOT TO 300 und Eingabe "r", um das zu fahren . . .

. . . Fein, funktioniert! Hier ein paar Übungen, damit Sie prüfen können, wie gut Sie begriffen haben. Lösungen am Kapitelende.

1 Das Quadrat soll nicht grün, sondern rot werden.
2 Es soll violett werden.
3 Machen Sie es blau und lassen Sie es blinken.
4 Es soll gelb sein und BRIGHT.
5 Setzen Sie es in Reihe 0, Spalte 1.
6 Setzen Sie es in Reihe 3, Spalte 7.

Für 1–4 brauchen Sie nur POKE 32000 entsprechend abzuändern; für 5 und 6 müssen Sie die Adresse 5800 zum richtigen Platz im Attributfile verändern.

Als nächstes füllen wir die ganze oberste Reihe grün aus. Wir brauchen eine Schleife mit einem Zähler, der auf 32 gesetzt wird und mit jedem Durchgang dekrementiert; wir prüfen, ob er auf Null steht, und wenn nicht, springen wir.

```
           LD HL, A-File        210058
           LD B, 32 dez         0620
Schleife:  LD (HL), 32 dez.     3620
           INC HL               23
           DEC B                05
           CP B                 B8
           JRNZ Schleife        20F9
```

Diesmal keine Datenbytes, also gleich zu "r", wenn die Option erscheint.

Wir können mühelos zwei, drei . . . Reihen erfassen, wenn wir den Zähler B größer machen. Wenn wir mit LD B, 64 dez [Ø64Øh] anfangen, werden *zwei* Reihen grün; [Ø66Ø] ergibt drei und so weiter bis [Ø6EØ], was sieben ergibt. Teilen wir B 20 hex (32 dez) mehr zu, heißt das, daß wir bei ØØ beginnen; die erste Dekrementierung von B bringt das auf 255 (keine Übertragsstellen gemerkt!), und das ist genauso, als hätten wir bei 256 für B angefangen. Und tatsächlich, wenn Sie die zweite Zeile oben verändern zu

	LD B,Ø	Ø6ØØ

füllen die obersten *acht* Reihen sich grün auf. Darüber hinaus können wir nicht gehen, jedenfalls nicht, wenn wir die ins B-Register geladenen Werte verändern, weil das ein 1 Byte-Register ist. Bevor wir das umgehen, wollen wir uns ein bißchen mehr anstrengen. B als Zähler zu verwenden, ist beim DJNZ-Befehl automatisch, und das spart ein paar Bytes. Ebenso gut geht es mit

	LD HL, A-File	21ØØ58
	LD B, Ø	Ø6ØØ
Schleife:	LD (HL), 32 dez	362Ø
	INC HL	23
	DJNZ Schleife	1ØFB

Probieren Sie das aus und sehen Sie selbst.

Farbe im Sofortverfahren

Um alle 24 Bildschirmreihen die Farbe wechseln zu lassen, können wir entweder BC als Zähler nehmen (und lassen ihn ab Ø3ØØ laufen) und ein CP C und ein weiteres JRNZ *Schleife* anfügen (um zu sehen, ob BC Null ist, müssen Sie B und C getrennt prüfen). Oder wir könnten das Obige mit einem anderen Register als Zähler in eine Schleife stellen. Oder, naive Burschen, die wir sind, könnten wir einfach drei Kopien des Programmes aneinanderzwängen und in verschiedenen Teilen des Attributfiles anfangen.

Versuchen wir das zuerst . . .

	LD HL, 58ØØ	21ØØ58
	LD B, Ø	Ø6ØØ
Schleife 1:	LD (HL), 32 dez	362Ø
	INC HL	23
	DJNZ Schleife 1	1ØFB
	LD HL, 59ØØ	21ØØ59
	LD B, Ø	Ø6ØØ

Schleife 2:	LD (HL), 32	362∅
	INC HL	23
	DJNZ Schleife 2	1∅FB
	LD HL, 5A∅∅	21∅∅5A
	LD B, ∅	∅6∅∅
Schleife 3:	LD (HL), 32 dez	362∅
	INC HL	23
	DJNZ Schleife 3	1∅FB

Fahren Sie mit ∅ Datenbytes; der ganze Bildschirm füllt sich sofort, der "Meldungsbereich" eingeschlossen. Um Letzteres zu verhindern, ändern Sie das dritte "LD B, ∅" ab zu "LD B, 192 dez" oder [∅6C∅], um beim dritten Durchgang nur sechs Reihen zu erhalten.

Ist Ihnen übrigens aufgefallen, daß man B nicht jedesmal auf Null zurücksetzen muß? Da ist es schon! Diese Zeilen könnten also weggelassen werden (außer der ersten, die immer notwendig ist und der dritten dann, wenn nur 22 Zeilen grün werden sollen).

Offensichtlich muß es einen schnelleren Weg geben, aber der obige hat eine angenehme Eigenschaft. Wir können die Farben unterwegs verändern. Machen Sie aus dem zweiten Vorkommen von [362∅] ein [361∅] und aus dem dritten [363∅] (am leichtesten geht das mit POKE 32∅16, 16: POKE 32∅26, 48), dann erhalten Sie drei Farbblöcke:

GRÜN
ROT
GELB

Eine Veränderung der Datenbytes ruft natürlich andere, ähnliche Effekte hervor.

Keine Ausreden. Die Zusatzschleife muß richtig funktionieren. Wir verwenden D als Schleifenzähler. Zur Abwechslung soll der Bildschirm violett werden.

	LD HL, A-File	21∅∅58
	LD D, ∅3	16∅3
äußere:	LD B, ∅	∅6∅∅
Schleife:	LD (HL), 24 dez	3618
	INC HL	23
	DJNZ Schleife	1∅FB
	DEC D	15
	CP D	BA
	JRNZ außen	2∅F5

Beachten Sie, daß wir HL in der Außenschleife nicht inkrementieren, weil sie schon fröhlich durch das Attributfile tickt; nur die Unfähigkeit von B allein, als Zähler zu dienen, muß ausgeglichen werden.

Wenn Sie sich ansehen wollen, um wieviel langsamer dieses Programm in BASIC läuft, schlagen Sie nach unter "Programmieren leicht gemacht", Kapitel 7.

FLASH ab und FLASH an

Die nächste Routine geht das Attributfile durch und schaltet jedes FLASH ab, läßt die Attribute aber sonst unverändert. Das FLASH-Bit im Attribut ist nun das am linken Ende, das heißt, das älteste Bit. Wir müssen es also auf Ø setzen, während wir den Rest unverändert lassen.

Ein Weg dazu ist der, eine Bit-*Maske* herzustellen. Wenn wir das Attributbyte in das A-Register setzen und per AND mit dem Bitmuster Ø1111111 verbinden, geht das erste Bit auf Ø und die anderen bleiben unverändert. Geht man davon aus, dann lautet der Code:

```
           LD BC, 768 dez    Ø1ØØØ3
           LD HL, A-File     21ØØ58
Schleife:  LD A, (HL)        7E
           AND 127 dez       E67F
           LD (HL), A        77
           INC HL            23
           DEC BC            ØB
           LD A, Ø           3EØØ
           CP B              B8
           JRNZ Schleife     2ØF5
           CP C              B9
           JRNZ Schleife     2ØF2
```

BC wird hier als Schleifenzähler verwendet: 768 ist natürlich die Länge des Attributfiles. Sowohl B als auch C müssen wegen dem Schleifenende auf Null geprüft werden.

Laden Sie das mit Ø Datenbytes, dann STOP. Wir brauchen etwas, womit wir das testen können. Fügen Sie Programmzeilen an

```
1ØØØ  CLS: For i = 1 TO 7Ø4: PRINT FLASH
      (INT (2 * RND)); CHR$ (65 + i – 2Ø * INT (i/2Ø));:
      NEXT i
1Ø1Ø  GO TO 3ØØ
```

Verwenden Sie GO TO 1ØØØ. Der Bildschirm füllt sich mit Buchstaben, von denen manche blinken. Geben Sie "r" für die Option ein, dann hört das Blinken auf.

Versuchen Sie INK und PAPER auf verschiedene Farben zu setzen und wiederholen Sie. Die Farben verändern sich nicht, wie Sie sehen.

Das umgekehrte Problem besteht darin, den ganzen Bildschirm auf FLASH zu setzen. Statt als Maske AND 127 dez zu verwenden, müssen wir OR 128 dez nehmen (128 ist binär 1ØØØØØØØ; OR verwandelt das erste Bit in 1 und läßt die anderen unverändert). Wir brauchen also genau dasselbe Problem, nur wird aus der Zeile

AND 127 dez E67F

die Zeile

OR 128 dez F6 8Ø

Sie können wie vorher Zeile 1ØØØ zum Testen nehmen. Was geschieht, wenn sie XOR mit 128 verwenden?

SET und RES

Die obigen Veränderungen im Bitmuster kann man auf direktere Weise erzielen. Der Befehl SET *setzt* ein beliebiges Bit in einem beliebigen Register auf 1, RES setzt es *zurück* (auf Ø).

Bits innerhalb des Bytes sind folgendermaßen geordnet:

7	6	5	4	3	2	1	Ø

so daß die größere Zahl auf ältere Bits verweist. Ein Befehl wie

SET 4, D

setzt Bit 4 des D-Registers auf 1, und

RES 6, E

setzt Bit 6 des E-Registers auf Null.

Statt also AND 127 dez zu verwenden, hätten wir nehmen können

RES 7, A CBBF

und statt OR 128 dez

SET 7, A CBFF

Diese besetzen dieselbe Zahl von Bytes, so daß keine Notwendigkeit besteht, die relativen Sprünge anzupassen; auch wenn Sie die Veränderungen vornehmen, laufen die Programme.

Spaltenscrolling

Jetzt wollen wir eine Routine schreiben, um eine Spalte von Attributen abzurollen – hier einmal Spalte 31 rechts außen. Denken Sie daran, daß die Attribute des Quadrats unmittelbar unter einem gegebenen an einer um 32 höheren Adresse zu Hause sind. Wir sollten also Indizieren mit einer Distanz von 32 verwenden. Es geht darum, die Attribute des unteren Quadrats in das D-Register und dann weiter in das obere Quadrat zu übertragen; gehen Sie für die ganze Spalte 21mal durch die Schleife. Das führt zu:

```
          LD BC, 32 dez     Ø12ØØØ
          LD IX, Start      DD211F58
          LD A, 21 dez      3E15
Schleife: LD D, (IX + 32)   DD562Ø
          LD (IX), D        DD72ØØ
          ADD IX, BC        DDØ9
          DEC A             3D
          CP B              B8
          JRNZ Schleife     2ØF4
```

Beachten Sie: Der Eintrag 1F im zweiten Befehl ist die Spaltennummer 31, die wir abrollen wollen, aber in Hex geschrieben. Bei einer anderen Spalte ändern Sie die Zahl ab.

Wir brauchen wieder etwas zum Testen. Laden Sie, geben Sie STOP und fügen Sie an

```
1ØØØ  FOR e=Ø TO 21: PRINT PAPER e – 7 * INT (e/7);
      "[32 × □]";: NEXT e
1Ø1Ø  GO TO 3ØØ
```

Nun GO TO 1ØØØ zu einem hübschen Streifenmuster zum Testen, und dann Option "r" für den Lauf.

Sie sehen, daß Spalte 31 um einen Leerraum nach oben rückt; wenn es mehr Plätze sein sollen, setzen Sie die Routine in BASIC in eine Schleife. Wenn es eine andere Spalte sein soll, ändern Sie das 1F ab.

Außerdem werden Sie bemerken, daß das untere Attribut in der Spalte unverändert bleibt. Um ein weißes Quadrat daraus zu machen (Attribut 56 = 7 * 8) fügen Sie dem Maschinencode folgende Zeile nach dem JRNZ an:

```
          LD (IX), 56 dez   DD36ØØ38
```

Eine Übung für Sie: Überlegen Sie, wie man einen kompletten Spaltenblock (sagen wir Spalten 5 bis 18) abrollt, indem man diese Routine innerhalb des Maschinencodes in eine Schleife setzt und jedesmal die Startadresse für IX inkrementiert.

Muster-Generator

Schließlich hier noch eine Routine, die FARBE IM SOFORTVERFAHREN mehr Pfiff verleiht und ausgefallene Muster liefert. Ich überlasse es Ihnen, sich darüber klarzuwerden, was sie im einzelnen leistet. Es gibt Ø Datenbytes.

```
LD B, Ø          Ø6ØØ
LD D, Ø          16ØØ
LD HL, 58ØØ      21ØØ58
LD (HL), D       72
INC HL           23
LD A, 7          3EØ7
ADD A, D         82
LD D, A          57
DJNZ ?           1ØF8
LD HL, 59ØØ      21ØØ59
LD (HL), D       72
INC HL           23
LD A, 7          3EØ7
ADD A, D         82
LD D, A          57
DJNZ ?           1ØF8
LD HL, 5AØØ      21ØØ5A
LD (HL), D       72
INC HL           23
LD A, 7          3EØ7
ADD A, D         82
LD D, A          57
DJNZ ?           1ØF8
```

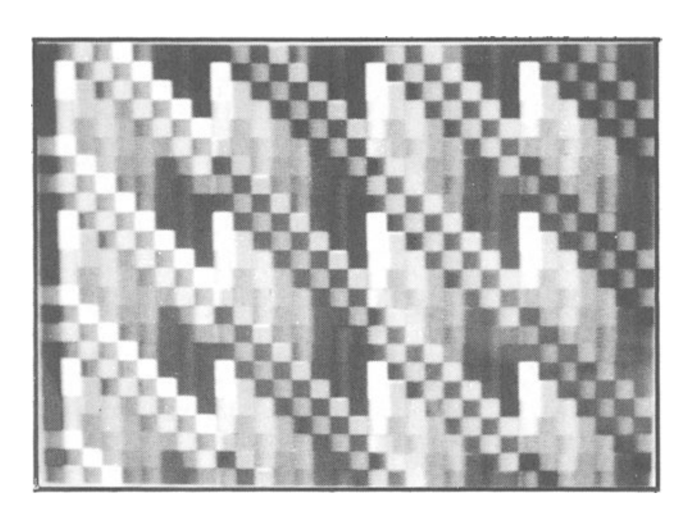


Abbildung 14.1
LD A,31 für dieses Würfelmuster. Was liefern die anderen Werte von A? 255 kann man ausprobieren.

Um andere Muster zu erzielen, verändern Sie die Zeile LD A, 7, um andere Zahlen nach A zu laden, etwa zu LD A, 8 oder LD A, 32 oder LD A, 31 (dezimal!) Statt Ø7 kann jede zweistellige Hexzahl verwendet werden, und in jedem der drei Fälle können andere Zahlen Verwendung finden.

Um den Bildschirm als erstes zu löschen, Stop (mit Option "a"), dann CLS, dann GO TO 3ØØ und Option "r".

Können Sie erkennen, was das Programm leistet? Und wie es das tut?

Lösungen

1 POKE 32ØØØ, 16

2 POKE 32ØØØ, 24

3 POKE 32ØØØ, 136

4 POKE 32ØØØ, 112

5 58ØØ zu 58Ø1 verändern 3AØØ7D32Ø158

6 58ØØ zu 5867 verändern 3AØØ7D326758

15 Das Displayfile

Das Displayfile ist wegen seiner komplizierten Struktur ein bißchen schwerer zu verwenden, aber die Resultate belohnen Hartnäckigkeit reichlich.

Vor allem muß man sich merken, daß das Displayfile bei 4ØØØ hex anfängt, 18ØØ Bytes (hex) lang und von Natur in drei Blöcken aufgebaut ist, die von 4ØØØ–47FF, 48ØØ–4FFF und 5ØØØ–57FF reichen.

Am besten verschafft man sich ein Gefühl dafür durch experimentieren; man nimmt Veränderungen am Displayfile-Teil des Speichers vor und sieht sich an, was dabei herauskommt.

Nehmen wir etwa an, Sie setzen in alle Adressen gleiche Werte. Was geschieht?

Verwenden wir, um konkret zu werden, das Byte 1Ø1Ø111Ø (binär) oder 174 (dez). Wir können HL dazu benützen, zur Adresse im Displayfile zu zeigen, und BC als Schleifenzähler. Dann haben wir

```
           LD HL, D-File       21ØØ4Ø
           LD BC, 18ØØ         Ø1ØØ18
Schleife:  LD (HL), 174 dez    36AE
           INC HL              23
           DEC BC              ØB
           CP B                B8
           JRNZ Schleife       2ØF9
           CP C                B9
           JRNZ Schleife       2ØF6
```

Man erhält ein Muster von senkrechten Streifen. Sie werden gebildet durch das Bitmuster der Zahl 174 in Binär, also 1Ø1Ø111Ø: ein schwarzer Streifen von Breite 1, dann ein weißer Streifen von Breite 1, dann ein schwarzer Streifen von Breite 1, dann ein weißer Streifen von Breite 1, dann ein schwarzer Streifen von Breite 3 und schließlich ein weißer Streifen von Breite 1. Das Ganze wiederholt sich 32mal von rechts nach links über den Bildschirm und zwar deshalb, weil jede Reihe des Displays 32mal mit demselben Bitmuster geladen wird; sie richten sich senkrecht aus und bilden Streifen.

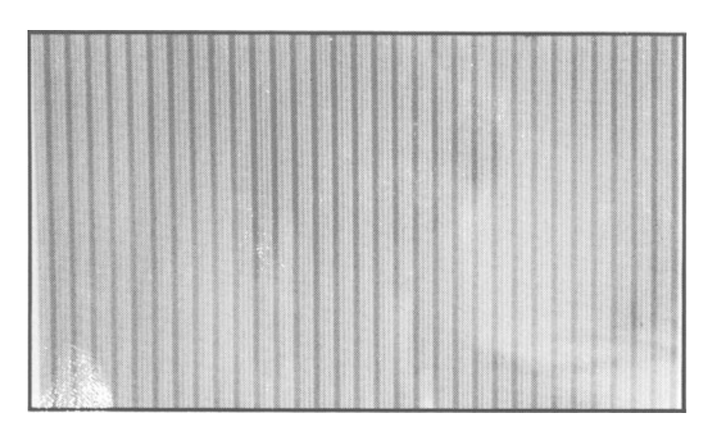


Abbildung 15.1
Identische Bytes liefern Wiederholungsstreifen.

Nehmen Sie statt 174 andere Zahlen und sehen Sie sich an, was für Muster erscheinen. Versuchen Sie vor allem 1, 15 und 17Ø, in Hex also Ø1, ØF und AA.

Horizontale Linien

Die nächste Routine zeichnet horizontale Linien (indem sie für bestimmte Displayreihen 255 in die Bytes setzt). Wie am Ende von Kapitel 14 löschen Sie am besten den Bildschirm mit CLS, bevor Sie Option "r" verwenden.

```
                LD A, 3             3EØ3
                LD B, Ø             Ø6ØØ
                LD HL, D-File       21ØØ4Ø
Schleife:       LD (HL), 255 dez    36FF
                INC HL              23
                DJNZ Schleife       1ØFB
                DEC A               3D
                CP B                B8
                JRZ Übersprung      28Ø8
                PUSH AF             F5
                LD A, 7             3EØ7
                ADD A, H            84
```

	LD H, A	67
	POP AF	F1
	JR loop	18EF
Übersprung:	(RET Befehl	C9)

Muster

Wenn Sie verschiedene Bytes in Teile des Displayfiles laden, können Sie recht auffallende Muster erzeugen. Das hier verwendet das C-Register, das bei ØØ beginnt und zu FF, FE, FD . . . und hinunter bis ØØ dekrementiert (dreimal im Ganzen), um das geladene Byte zu definieren. Bei sorgfältiger Analyse des Displays können Sie also sogar die Anordnung im Displayfile erkennen. In der Praxis bringt das Muster diese Analyse durcheinander, falls Sie die Antwort nicht schon kennen . . .

	LD BC, 768 dez	Ø1ØØ18
	LD HL, D-File	21ØØ4Ø
Schleife:	LD (HL), C	71
	INC HL	23
	DEC BC	ØB
	CP B	B8
	JRNZ Schleife	2ØFA
	CP C	B9
	JRNZ Schleife	2ØF7

Versuchen Sie als Variation zum Thema die Schleifenzeile so zu verändern, daß statt dessen das B-Register geladen wird:

Schleife:	LD (HL), B	7Ø

Das liefert ein anderes Muster. Es geht 256mal durch die Schleife, um den Wert von B zu ändern (als dem älteren Byte des Schleifenzählers), was acht Hochauflösungszeilen des Bildschirms entspricht. Das Muster macht deutlich, daß von den oberen acht Reihen jede verschiedene Werte von B hat, so daß die ersten 256 Bytes *nicht* den Reihen 175–168 entsprechen (wie sie es tun würden, wenn man die Reihen von oben bis unten der Folge nach vornehmen würde); und die Art, wie das Muster sich innerhalb eines Blocks von 8 Reihen wiederholt, zeigt, daß die ersten 256 Bytes, wie ich oben schon erklärt habe, tatsächlich die Zeilen 175, 167, 159 etc. enthalten. Die Struktur aus drei Blöcken ist in diesem Muster wie im vorigen sehr deutlich.

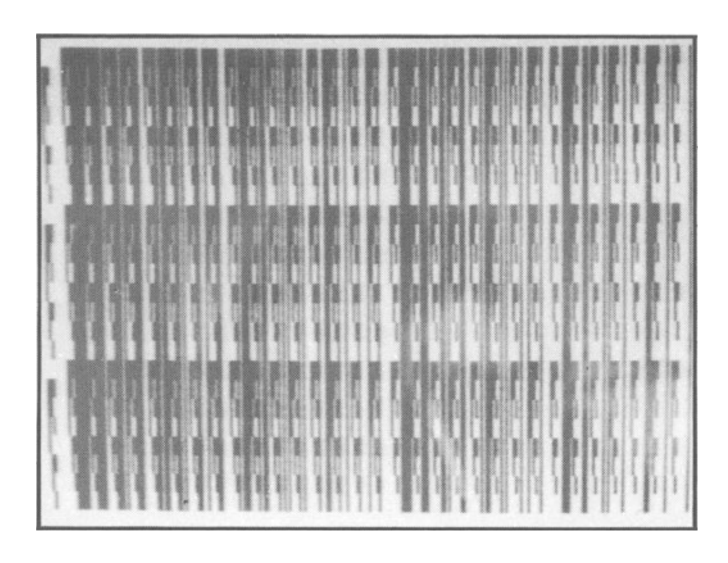


Abbildung 15.2
Variable Bytes liefern fremdartige Muster. Sehen Sie die drei Bildschirmblöcke? Können Sie dafür verantwortliche Binärzahlen erkennen?

Solche Routinen verleihen uns ein Gefühl, daß wir das Displayfile zu interessanten Dingen bewegen können, wenn wir das wirklich wollen, aber sie sind nicht direkt *nützlich*, außer daß sie Selbstvertrauen schaffen.

Schraffieren

Diese Routine hat den Vorteil, daß die detaillierte Struktur des Displayfiles nicht wichtig ist. Sie geht das File durch und ersetzt jedes Nullbyte ØØØØØØØØ durch das Byte 1Ø1Ø1Ø1Ø (AA hex). Die Wirkung ist die, daß die leeren Stellen des Bildschirms mit dünnen senkrechten Nadelstreifen überzogen werden.

```
             LD HL, D-File      21ØØ4Ø
             LD BC, 768 dez     Ø1ØØ18
             LD A, Ø            3EØØ
Schleife:    CP A, (HL)         BE
             JRNZ Übersprung    2ØØ2
             LD (HL), AA hex    36AA
Übersprung:  INC HL             23
             DEC BC             ØB
             CP B               B8
```

JRNZ Schleife	2ØF6
CP C	B9
JRNZ Schleife	2ØF3

Der Befehl LD A, Ø ist eigentlich nicht notwendig: A enthält immer Null, wenn es nicht mit etwas anderem geladen wurde. Aber die Art, wie das Programm funktioniert, wird dadurch ein bißchen klarer. Vergewissern Sie sich, daß er nicht gebraucht wird, indem Sie ihn weglassen. (Am schnellsten geht das mit POKE 32ØØ6, Ø, das die Zeile zu ØØØØ verwandelt. Der Code ØØ bedeutet NOP – NoOPeration – (keine Operation) und leistet nichts, außer Zeit zu vergeuden. Dadurch ist er ideal für das versuchsweise Löschen eines Befehls, weil die anderen Hexcodes im Speicher nicht umherbewegt werden müssen.)

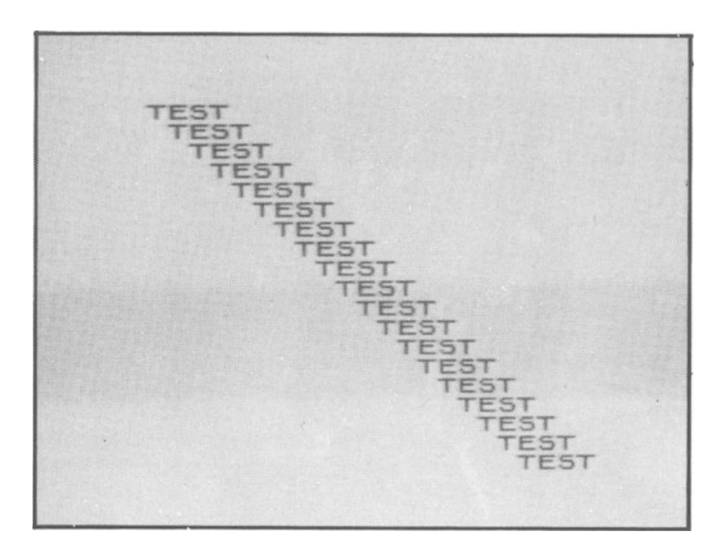
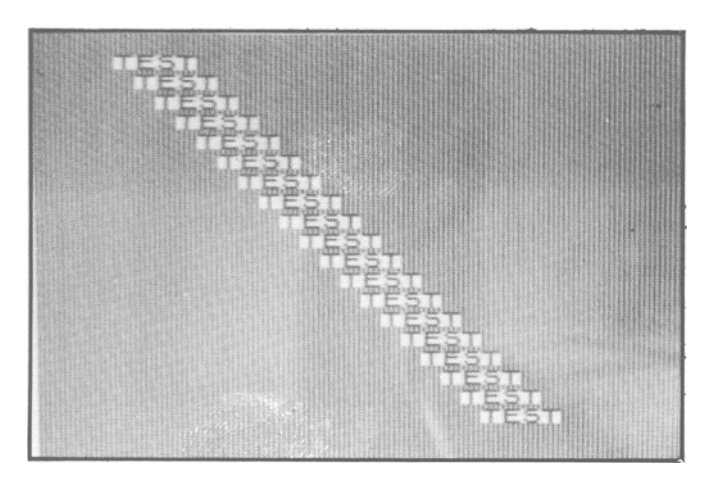


Abbildung 15.3
Vor dem Schraffieren . . .

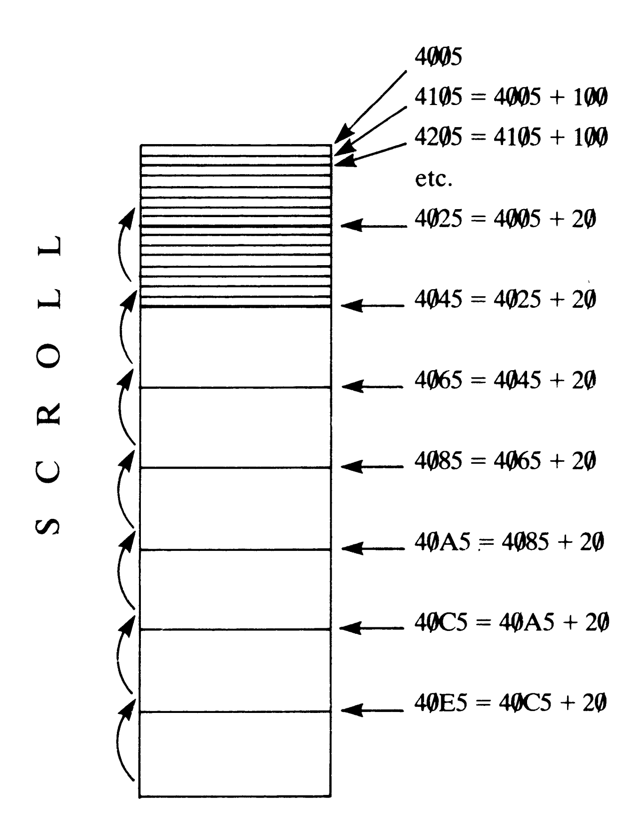


Abildung 15.4
. . . und danach . . .

Einzelspalten-Scrolling

Jetzt zu einer komplizierteren Sache: eine Einzelspalte des Bildschirms wegrollen. Um das Problem ein wenig zu vereinfachen, will ich nur in einem von drei Blöcken scrollen, das heißt, einen 8 Reihen-Abschnitt der Spalte. Ich nehme Block 1, Spalte 5.

Spalte 5 sieht so aus: Sie besteht aus 8 Zeilen-Abschnitten, von denen jeder einem Zeichen (und einer PRINT AT-Zelle entspricht, und jede Zeile wird an der gezeigten Adresse gespeichert).



Um Verwirrung zu vermeiden, nenne ich jedes Quadrat von 8 Zeilen eine *Reihe* (was sie für eine PRINT AT-Anweisung ist) und jede Einzelzeile eine *Zeile*. Die oberste Zeile von Reihe ∅ ist dann in Adresse 4∅∅5 (hex) im Displayfile enthalten. Hinzufügen von 32 dezimal (2∅ hex) liefert die oberste Zeile von Reihe 1, und so weiter hinab zu Reihe 7.

Um die zweite Zeile in Reihe ∅ zu erhalten, müssen wir 256 (dez) oder 1∅∅ (hex) den ursprünglichen 4∅∅5 hinzufügen. Das ergibt

4∅∅5 + ∅1∅∅ = 41∅5

Durch weiteres Hinzufügen von 2∅ (hex) erhalten wir die zweiten Zeilen der anderen sieben Reihen. Dann gehen wir weiter zu den dritten Zeilen . . . bis wir zuletzt mit den achten Zeilen abschließen.

Das ist der Aufbau der Spalte (Block 1). Um ein Byte um eine Reihe hochzurollen, stellen wir es in die Adresse 32 Plätze früher (dezimal). Wir wollen die oberste Reihe ganz verlieren und die siebte Reihe am Ende leer lassen.

Das ist ganz ähnlich wie das Spaltenscrolling von Attributen, das ich vorher geschrieben hatte, nur muß jetzt achtmal durch die Schleife gegangen werden, damit alle acht Zeilen in einem Quadrat bewegt werden. (Eine Routine,

die nur das oberste Achtel jedes Zeichens abrollen würde, wäre nicht sehr nutzvoll – zumal sie alle leer sind!)

Wir müssen also mit einer Distanz von Ø oder 32 wie vorher indizieren.

In der schwachen Hoffnung, das alles klarer zu machen, lege ich zuerst eine BASIC-Version der Routine vor.

```
2ØØØ  LET ix = 256 * 64 + 5
2Ø1Ø  FOR l = 1 TO 8
2Ø2Ø  FOR a = 1 TO 7
2Ø3Ø  LET b = PEEK (ix + 32)
2Ø4Ø  POKE ix, b
2Ø5Ø  LET ix = ix + 32
2Ø6Ø  NEXT a
2Ø7Ø  POKE ix, Ø
2Ø8Ø  LET ix = ix + 32
2Ø9Ø  NEXT l
```

Hier habe ich Variable in Kleinbuchstaben wie ix als Entsprechung zu den Großbuchstabenregistern wie IX verwendet. Die a-Schleife schiebt alle oberen Zeilen hinauf; Zeile 2Ø7Ø gibt mit POKE den Leerraum für die siebte Reihe ein, und die l-Schleife wiederholt den Prozeß für die 2., 3. bis 8. Zeile in jedem Zeichen.

Stellen Sie etwas zum Testen auf und verwenden Sie LIST 2ØØØ: GO TO 2ØØØ.

Wenn Sie das ausprobieren, funktioniert es, aber es geht ziemlich langsam. Die Zeichen "tröpfeln" sichtbar nach oben. Aber wir sind wenigstens auf der richtigen Spur. (Das ist ein nützlicher Debugging-Trick: zuerst eine BASIC-Version der Routine schreiben, um zu prüfen, ob die Idee vernünftig ist; *sie* von Fehlern befreien, dann erst zu Maschinencode übergehen.)

Die Umwandlung von BASIC in Maschinencode ergibt:

```
             LD DE, ØØ2Ø       112ØØØ
             LD IX, 4ØØ5       DD21Ø54Ø
             LD L, Ø8          2EØ8
Schleife 1:  LD A, Ø7          3EØ7
Schleife 2:  LD B, (IX + 2Ø)   DD462Ø
             LD (IX), B        DD7ØØØ
             ADD IX, DE        DD19
             DEC A             3D
             CP D              BA
```

JRNZ Schleife	2ØF4
LD (IX), ØØ	DD36ØØØØ
ADD IX, DE	DD19
DEC L	2D
PUSH AF	F5
LD A, Ø	3EØØ
CP L	BD
POP AF	F1
JRNZ Schleife	2ØE4

Um das zu testen, fügen Sie in BASIC hinzu:

```
1ØØØ  FOR i = Ø TO 21: FOR j = Ø TO 31:
      PRINT CHR$ (65 + i);: NEXT j: NEXT i
```

Dann GO TO 1ØØØ und die Routine fahren (entweder über GO TO 3ØØ oder durch ein direktes RANDOMIZE USR 32ØØØ). Der obere Block von Spalte 5 rollt in der Tat nach oben – mit gemessenem Schritt. Wenn Sie den Maschinencode in BASIC in eine Schleife stellen, erhalten Sie ein annehmbar schnelles Scrolling

```
3ØØØ  FOR k = 1 TO 8:  RANDOMIZE USR 32ØØØ: NEXT k
```

Wenn Sie das in Maschinencode in eine Schleife tun, läuft es so schnell, daß Sie die Zeichen kaum verschwinden sehen.

Vielspalten-Scrolling

Wenn das Vorherige läuft, fällt es leicht, es zu einer Routine zu erweitern, um innerhalb eines Blocks einen ganzen Abschnitt von Spalten abzurollen. Dazu sind aber noch Vorbereitungen nötig.

Setzen Sie vier Datenbytes in 7DØØ–7DØ3 beiseite. Diese enthalten die Startspalte, den Block, die Breite des abzurollenden Abschnitts und eine Blindnull, die wegen der Säuberlichkeit gebraucht wird.

7DØØ	Spalte	(etwa Ø5 für Spalte 5 Start)
7DØ1	Block	(4Ø, 48 oder 50 für die drei Blöcke)
7DØ2	Breite	(etwa 11 für eine Breite von 17 [dez])
7DØ3	ØØ	(Blindnull)

Der Maschinencode, der bei 7DØ4 beginnt (aufgerufen von RANDOMIZE USR 32ØØ4) sieht so aus; beachten Sie, daß der mittlere Teil nur eine Wiederholung dessen ist, was wir oben entwickelt haben.

```
             LD BC, (7DØ2)       ED4BØ27D
             LD DE, ØØ2Ø         112ØØØ
             LD IX, (7DØØ)       DD2AØØ7D
Schleife 3:  PUSH IX             DDE5
             LD L, Ø8            2EØ8
Schleife 2:  LD A, Ø7            3EØ7
Schleife 1:  LD B, (IX + 2Ø)     DD462Ø
             LD (IX), B          DD7ØØØ
             ADD IX, DE          DD19
             DEC A               3D
             CP D                BA
             JRNZ Schleife 1     2ØF4
             LD (IX), ØØ         DD36ØØØØ
             ADD IX, DE          DD 19
             DEC L               2D
             PUSH AF             F5
             LD A, Ø             3EØØ
             CP L                BD
             POP AF              F1
             JRNZ Schleife 2     2ØE4
             POP IX              DDE1
             DEC C               ØD
             PUSH AF             F5
             LD A, Ø             3EØØ
             CP C                B9
             POP AF              F1
             JRZ Übersprung      28Ø4
             INC IX              DD23
             JR Schleife 3       18D2
Übersprung:  (RET                C9)
```

Um das zu nutzen, geben Sie die Datenbytes durch POKE mit den entsprechenden Zahlen ein (z. B. 5, 64, 17, Ø – wohlgemerkt, POKE arbeitet mit Dezimal, nicht mit Hex!) Dann GO TO 1ØØØ mit dem kleinen Testprogramm oben, um etwas zu setzen, das man abrollen kann. Dann entweder GO TO 3ØØ mit Option "r" oder RANDOMIZE USR 32ØØ4. Läuft schon nach oben!

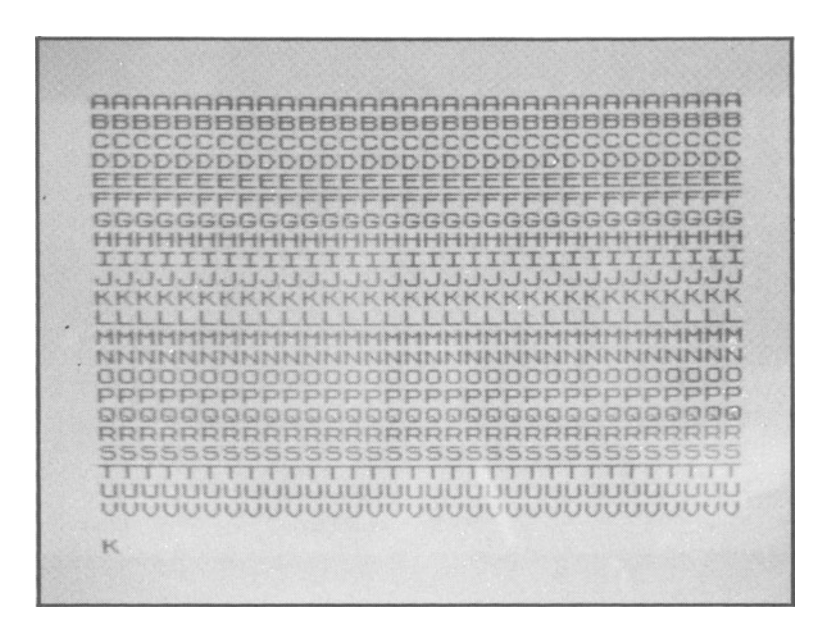


Abbildung 15.5
Vor dem Abrollen eines Bildschirmteils . . .

Versuchen Sie, in BASIC mit der Schleife zu arbeiten. Das ist sehr effektiv. Sie können eine Scrollroutine wie diese bei Block 2 verwenden, etwa um ein sehr hübsches Programm für einen Geldspielautomaten zu verfassen . . .

Eine letzte Bemerkung. Als ich die Routine das erstemal ausprobierte, kam ich bei den letzten Zeilen ein bißchen durcheinander; ich setzte das POP AF nach dem JRZ *Übersprung*. Die Folge war, daß

1 der Bildschirm sehr schön abrollte,

aber

2 die Fehlermeldung C: Nonsense in Basic Ø:1 kam

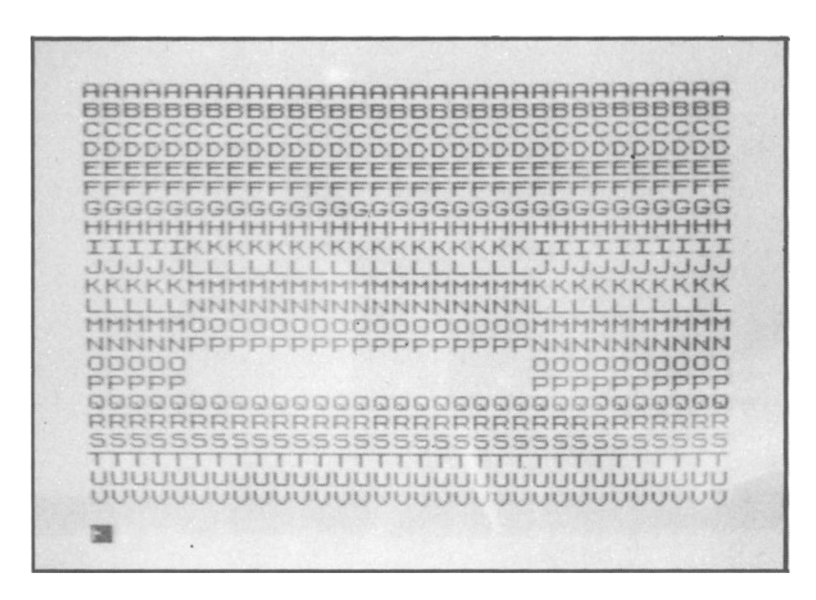


Abbildung 15.6
. . . und das Ergebnis, nach zweimaligem Abrollen.

Das könnte Ihnen auch passieren. Es ist ein Zeichen dafür, daß mit dem *Stapel* etwas schiefgegangen ist. Und beim Code in dieser Reihenfolge springt der letzte JR *Übersprung* über den POP AF-Befehl. Wenn der Spectrum dann zu BASIC zurückzukehren versucht, sitzt in der Rücksprungadresse auf dem Stapel (eine 2 Byte-Adresse) dieses Restbit vom AF-Register, und das System gerät völlig in Verwirrung. Es ist leider zu einfach, aus einer Maschinencode-Routine mit durch POP noch nicht entfernten Stapelbits herauszugehen, aber keine sehr gute Idee.

16 Mehr über Flaggen

In Maschinencode gibt es keine FOR/NEXT-Schleifenbefehle oder IF/THEN-Bedingungen. Um sie zu erhalten, muß man die Flaggen verwenden. Als ein Beispiel hier ein Programm zur Zeilenumnumerierung.

Ich habe bis jetzt versucht, die technischen Details des F-Registers zu meiden, aber bei jedem bedingten Sprung haben wir stillschweigend die Flaggen benutzt. Sich genauer damit zu befassen, lohnt also. Ich will auf keinen Fall alle schrecklichen Einzelheiten aufführen; wie das F-Register genau funktioniert, ist eine der kompliziertesten Eigenheiten des Z8∅. Sich ein bißchen genauer damit abzugeben, ist aber auf keinen Fall eine schlechte Idee.

Das F-Register hat Platz für acht Bits, verwendet aber nur sechs davon. Das sind:

C	Übertragsflagge
Z	Nullflagge
S	Vorzeichenflagge
P/V	Paritäts/Überlaufflagge
H	Halbübertrags-Flagge
N	Subtrahierflagge

Im Register sind sie folgendermaßen angeordnet:

S	Z	X	H	X	P/V	N	C

wobei "X" für "nicht verwendet" steht.

Die Übertragsflagge wird vor allem beeinflußt durch Additions-, Subtraktions-, Rotations- und Schiebebefehle.

Die Nullflagge wird von fast allem beeinflußt! Grob gesprochen: Wenn irgend etwas außer LD, INC, DEC den Inhalt von A verändert, wird die Nullflagge gesetzt (auf 1), falls A Null ist, sonst zurückgesetzt (auf ∅). BIT setzt die Flagge, wenn das angegebene Bit Null ist. CP setzt die Flagge oder setzt sie zurück nach dem Ergebnis eines Vergleichs.

Die Vorzeichenflagge speichert das Vorzeichenbit des Resultats jener Operation, die zuletzt ausgeführt worden ist: 1 für negativ, ∅ für positiv.

Die P/V-Flagge wirkt in einer Beziehung für Arithmetik und in einer anderen für Logik. Bei Arithmetik wird sie auf 1 gesetzt, wenn in Zweierkomplement-Arithmetik ein *Überlauf* vorliegt (Beispiel: Wenn die Summe zweier positiver Zahlen am Ende des Akkumulators überläuft und zu scheinbar negativen Ergebnissen führt). Bei einer logischen Operation wird sie auf ∅ gesetzt,

wenn das Byte in A eine *gleiche* Zahl von Bits besitzt, die 1 entsprechen, und auf 1, wenn die Zahl der 1 entsprechenden Bits *ungerade* ist. Diese gerade/ungerade-Eigenschaft der Bits nennt man die *Parität* des Bytes. Beispiel:

A enthält Ø11Ø11ØØ: gerade Parität. P/V-Flagge Ø

A enthält 1ØØ1ØØØ1: ungerade Parität. P/V-Flagge 1.

Die H- und N-Flaggen werden nur für binär codierte Dezimalberechnungen verwendet; daß Sie diese bei einem Spectrum brauchen, ist unwahrscheinlich. Sie sind meist dazu da, andere Ausgabegeräte zu betreiben. Die Art, wie C- und Z-Flagge durch verschiedene Operationen beeinflußt werden, ist in Anhang 5 aufgeführt. Das sind für den Anfänger die nützlichsten Flaggen.

Wir haben die Nullflagge in einer Reihe von Routinen verwendet, beispielsweise jedesmal bei JRZ, JRNZ oder DJNZ. In der Regel geht diesen bedingten Sprüngen ein *Vergleichs*befehl voraus, etwa

```
CP B
```

der die *Flaggen so setzt*, als wäre er der Befehl SUB A, B; er verändert aber den Inhalt von A nicht. Wenn A und B dasselbe Byte enthalten, würde diese Subtraktion Null ergeben, und die Nullflagge wird *gesetzt* (auf 1). Wenn A und B verschieden sind, wird die Nullflagge zurückgesetzt.

Die Übertragsflagge haben wir aber wenig verwendet. Sie ist besonders nützlich als Schleifenzähler, wenn man nicht genau weiß, ob man das genaue Schleifenende trifft; man könnte darüber hinausschießen.

Konkret: Angenommen, Sie wollen wissen, ob die Zahl im HL-Register größer ist als die im BC-Register. Der Befehl

```
SBC HL, BC        ED42
```

setzt die Übertragsflagge, wenn BC größer als HL, und setzt sie zurück, wenn BC kleiner oder gleich HL. Schließen Sie einen bedingten Sprung

```
JR C irgendwohin   38
```

an, und der Sprung findet statt, wenn BC größer als HL.
Fahren Sie fort mit

```
JR NC irgendwohin 3Ø??
```

und der Sprung findet statt, wenn BC gleich oder kleiner HL. (Für die Bestimmung der Größe werden alle Zahlen als positiv behandelt – keine Zweierkomplement-Arithmetik!)

Zeilen umnumerieren

Ich kann das darstellen anhand einer einfachen Routine zur Zeilenumnumerierung. Das läuft nur durch den BASIC-Programmbereich und verändert die Zeilennummern zur regulären Folge 1Ø, 2Ø, 3Ø . . . in Zehnerschritten aufwärts bis zum Ende. (Ein raffinierteres BASIC-Programm steht in "Weitere Kniffe und

Programme mit dem ZX Spectrum"* und dieses könnte auch noch verbessert werden; hier wollte ich etwas leicht Verständliches.)

Dazu muß man wissen, wie die BASIC-Zeilen gespeichert werden. Im oben genannten Buch steht das zwar auch, aber ich will es hier noch einmal angeben.

Das BASIC-Programm wird zwischen den Adressen in PROG und VARS gespeichert. Ohne Diskettenlaufwerke ist PROG 23755. Jede Zeile wird in der Form

NS	NJ	LJ	LS	Code für Zeile	ENTER

festgehalten. NS und NJ sind hier die älteren und jüngeren Bytes der Zeilennummer, LJ und LS die jüngeren und älteren Bytes der Zahl der Zeichen in der Zeile, das ENTER eingeschlossen (aber nicht die ersten vier Bytes NS, NJ, LJ, LS). Beachten Sie, daß in der Zeilennummer das ältere Bit voransteht – *kein* Druckfehler.

Beispielsweise wird das Testprogramm

```
1  REM 12345678901
5  PRINT a
```

im Speicher so festgehalten:

* Von Ian Stewart und Robin Jones, erschienen im Birkhäuser Verlag, Basel (Schweiz).

Adresse	Inhalt	Hinweise
23755	∅	älteres Byte der ersten Zeilennummer
23756	1	jüngeres Byte der ersten Zeilennummer
23757	13	jüngeres Byte der Zeilenlänge
23758	∅	älteres Byte der Zeilenlänge
23759	234	Code für REM
2376∅	49	Code für 1
23761	5∅	Code für 2
23762	51	Code für 3
23763	52	Code für 4
23764	53	Code für 5
23765	54	Code für 6
23766	55	Code für 7
23767	56	Code für 8
23768	57	Code für 9
23769	48	Code für ∅
2377∅	49	Code für 1
23771	13	Code für ENTER
23772	∅	älteres Byte der 2. Zeilennummer
23773	5	jüngeres Byte der 2. Zeilennummer
23774	3	jüngeres Byte der Zeilenlänge
23775	∅	älteres Byte der Zeilenlänge
23776	245	Code für PRINT
23777	97	Code für a
23778	13	Code für ENTER
23779		Beginn des Bereichs VARIABLES

Sie können das mit der PEEK-Routine in "ZX Spectrum – Programmieren leicht gemacht", S. 124 nachprüfen. Oder ein eigenes Programm schreiben. Versuchen Sie's!

Damit ist der naheliegende Weg, Zeilen umzunumerieren, der, den Programmbereich durchzugehen und nach ENTER-Bytes zu suchen (Code 13). Die beiden nächsten Bytes sind Zeilennummern; verändern Sie sie.

Schön. Geht aber nicht. Der Grund: Das Byte 13 kann auch anderswo vorkommen, vor allem in den Zeilenlängenbytes. In Adresse 23757 oben ist das auch der Fall (deshalb habe ich dieses Testprogramm gewählt). Sie *können* das umgehen, aber es ist umständlich. (Bei einem ZX81 ist das entsprechende Byte NEWLINE, Code 118). In der Regel umfassen Zeilen keine 118 Zeichen, also ist das dort kein solches Problem.)

Außerdem gibt es einen besseren Weg. Die Zeilenlängenbytes *sagen* Ihnen genau, wie weit Sie gehen müssen, um die nächste Zeilennummer zu erreichen. Warum lange nach ENTER suchen?

Und das führt zu folgendem Code.

Das Programm

Es geht darum, bei den ersten beiden Bytes des Programmbereichs, erste Zeilennummer, zu beginnen, diese umzunumerieren, die nächsten beiden Zeilenlängenbytes zu benutzen, um zur nächsten Zeilennummer zu springen, und das Ganze zu wiederholen, bis Sie zum VARS-Bereich gelangen.

Als erstes müssen wir die Adressen speichern. Wir verwenden BC, um den VARS-Wert aufzunehmen (wo wir anhalten), HL für PROG (wo wir anfangen), und DE für die neue Zeilennummer. Diese werden initialisiert:

```
                LD BC, (VARS)   ED4B4B5C
                LD HL, (PROG)   2A535C
                LD DE, 1Ø dez   11ØAØØ
```

Wegen der Systemvariablenadressen PROG und VARS siehe Anhang 3.

Als nächstes wollen wir feststellen, ob das HL-Register (das wir als durch den Programmbereich laufenden Zeiger verwenden wollen, weil HL für indirekte Steuerung ausgezeichnet geeignet ist) den Wert im BC-Register übersteigt. Wenn ja, halten wir an. *Da* kommt die Übertragsflagge an die Reihe:

```
prüfen:         PUSH HL         E5
                SBC HL, BC      ED42
                POP HL          E1
                JRNC Ende       3Ø15
```

(Die letzte 15 kann in Wirklichkeit nicht errechnet werden, bis das ganze Programm geschrieben ist, aber sie kommt am Ende heraus.)

Dann die eigentliche Umnumerierung:

```
umnumerieren:   LD (HL), D      72
                INC HL          23
                LD (HL), E      73
```

Danach müssen wir die nächsten beiden Bytes lesen, um die Zeilenlänge zu finden. Das Resultat müssen wir irgendwo speichern. Das HL-Register wird für Berechnungen gebraucht, also bietet sich das BC-Register an. Leider ist es in Gebrauch. Das können wir aber umgehen, indem wir den laufenden Wert speichern, um ihn später zurückholen zu können:

```
PUSH BC       C5
INC HL        23
LD C, (HL)    4E
INC HL        23
LD B, (HL)    46
```

Wir verschieben HL zur nächsten Zeilennummer hinauf

```
ADD HL, BC    Ø9
INC HL        23
```

und erhalten den alten BC-Wert wieder

```
POP BC        C1
```

Schließlich wollen wir DE auf den richtigen Wert für die nächste Zeile setzen, indem wir 1Ø hinzufügen. Dazu müssen erneut Register auf den Stapel verschoben werden:

```
PUSH HL       E5
LD HL, 1Ø dez 21ØAØØ
ADD HL, DE    19
LD D, H       54
LD E, L       5D
POP HL        E1
```

Nun mit der Schleife zurück zum Schritt *prüfen*, um zu wiederholen, außer wir haben HL zu groß gemacht:

```
JR prüfen     18E5
```

Zuletzt erscheint der abschließende RET-Befehl an der Stelle, die oben als *Ende* bezeichnet wurde.

```
   5 CLEAR 31999
   6 POKE 23609,50
  10 PRINT "base address ";
  15 DIM h$(2)
  20 INPUT b: PRINT b
  27 PRINT "No. of data bytes ";
  40 INPUT d: PRINT d
  50 FOR i=0 TO d-1
  55 FOR i=0 TO d-1
  60 POKE b+i,0
  70 NEXT i
  80 LET a=b+d
  87 PRINT "CODE:"
 100 INPUT c$
 106 IF c$="s" THEN GO TO 200
 110 IF c$="s" THEN GO TO 200
 120 PRINT c$+" ";
 128 LET hs=CODE c$(1)-48-39*(c$
(1)>"9")
 140 LET hj=CODE c$(2)-48-39*(c$
(2)>"9")
 150 POKE a,16*hs+hj

BEFORE        L
```

Abbildung 16.1
Vor der Umnumerierung . . .

```
  10 CLEAR 31999
  20 POKE 23609,50
  30 PRINT "base address ";
  40 DIM h$(2)
  50 INPUT b: PRINT b
  60 PRINT "No. of data bytes ";
  70 INPUT d: PRINT d
  80 FOR i=0 TO d-1
  90 FOR i=0 TO d-1
 100 POKE b+i,0
 110 NEXT i
 120 LET a=b+d
 130 PRINT "CODE:"
 140 INPUT c$
 150 IF c$="s" THEN GO TO 200
 160 IF c$="s" THEN GO TO 200
 170 PRINT c$+" ";
 180 LET hs=CODE c$(1)-48-39*(c$
(1)>"9")
 190 LET hj=CODE c$(2)-48-39*(c$
(2)>"9")
 200 POKE a,16*hs+hj

AFTER         L
```

Abbildung 16.2
. . . und nachher. Die GO TOs sind nicht verändert.

Laden Sie die ganze Routine in der gegebenen Reihenfolge (wie üblich mit dem abschließenden RET). Welches Programm Sie im BASIC-Bereich jetzt auch haben mögen, es wird umnumeriert, wenn Sie eingeben

RANDOMIZE USR 32ØØØ

Natürlich ändern sich nur die Zeilennummer – GO TO- und GO SUB-Nummern stimmen nicht mehr überein. Aber das Prinzip ist klar.

17 Blocksuche und Blockübertragung

Es gibt einige sehr wirksame Befehle, die auf einen Schlag ganze Speicherblöcke bewältigen.

Einige der Routinen, die ich oben gezeigt habe, sind nicht auf die leistungsfähigste Weise geschrieben. Es ging darum, für den Anfang möglichst einfach vorzugehen, und dafür will ich mich nicht entschuldigen. Maschinencode zu bewältigen, ist nicht gerade das Einfachste, und wenn man alle Merkmale auf einmal aufführt, stiftet das nur Verwirrung. Wenn Sie sich aber andere Bücher über Z8Ø-Maschinencode oder Listings in Zeitschriften angesehen haben, werden Sie sich vielleicht fragen, warum ich mich manchmal so dumm angestellt habe. Dieses und das nächste Kapitel machen das vielleicht wieder gut.

Blocksuche

Erstens gibt es ein paar sehr wirksame Befehle, die einen ganzen Speicherblock absuchen. Ich nehme CPDR, abgekürzt für "compare, decrement and repeat" (vergleiche, dekrementiere und wiederhole) als Beispiel.

Wenn wir einen Code dieser Art schreiben:

```
        LD BC, Ø1ØØ    Ø1ØØØ1
        LD HL, 5ØØØ    21ØØ5Ø
        LD A, Ø5       3EØ5
        CPDR           EDB9
dann:   --
```

geschieht Folgendes:

Wenn der CPDR-Befehl auftaucht, wird der Wert im A-Register mit dem Inhalt des Bytes verglichen, auf das HL zeigt. Sind sie gleich, geht die Steuerung an *dann* über. Sind sie es nicht, werden BC und HL um 1 dekrementiert, und das "vergleiche" wird wiederholt, bis eine Entsprechung gefunden wird oder bis BC Null enthält. Mit anderen Worten: Diese vier Befehle sagen: "Finde das erste Vorkommen eines Bytes, das Ø5 enthält, von Adresse 5ØØØ (hex) bis hinunter zu Adresse 4FØØ und laß HL darauf zeigen. Wenn keines vorhanden ist, stell BC auf Null."

Mein erstes Beispiel für die Verwendung von Sprüngen, eine kleine "Vergleiche"-Schleife, hätte also viel leichter bewältigt werden können. Dann hätte es aber eben keine Sprünge vorgeführt!

Es gibt einen Begleitbefehl CPIR, der statt dessen das HL-Register inkrementiert, sonst aber genauso wirkt. Damit wird also ein Speicherblock vom anderen Ende her abgesucht.

Blockübertragung

Die *Blockübertragungs*-Befehle LDIR und LDDR verschieben Datenblöcke innerhalb des Speichers. Nehmen wir LDIR. Sie

1 laden HL mit der Adresse des ersten Bytes, das übertragen werden soll,
2 laden DE mit der Adresse des ersten Zielbytes,
3 laden BC mit der Zahl der zu verschiebenden Bytes.

LDIR überträgt dann das erste Byte, inkrementiert HL und DE, dekrementiert BC und setzt das fort, bis BC bei Ø ist. Ein wichtiger Punkt, den man sich merken muß: Der abschließende Schritt betrifft die Inkrementierung von HL und DE, führt aber keine Übertragung durch. Am Ende zeigt HL also auf das Byte unmittelbar nach dem Übertragungsbereich, DE auf das obere Ende des Übertragungsbereichs. Vergleiche die Routinen für seitliches Abrollen weiter unten.

LDDR ist ähnlich, dekrementiert aber HL und DE (und dekrementiert weiter BC wie bei LDIR – BC ist nur ein Zähler).

Vorausbestimmte Attribute

Ein Weg, Blockübertragung zu nutzen, ist der, im RAM ein "falsches" Attributfile aufzubauen und es in das wahre Attributfile zu übertragen – wodurch schlagartig alle Attribute zu einem neuen, vorausbestimmten Muster verändert werden. Dazu brauchen Sie 704 Datenbytes für die neuen Attribute. Das erfordert eine Änderung im LADER-Programm. Zeile 1Ø muß nun lauten:

```
1Ø  CLEAR 31599
```

wodurch genug Platz bleibt. Dann RUN und die Mitteilung, daß 7Ø4 Datenbytes vorhanden sind. Der Code sieht so aus:

```
LD HL, 316ØØ dez   217Ø7B
LD, DE, A-File     11ØØ58
LD BC, 7Ø4 dez     Ø1CØØ2
LDIR               EDBØ
```

316ØØ (7B7Ø hex) ist der Beginn des neuen Datenbereichs. Die Routine selbst wird in die Startadresse 323Ø4 geladen.

Jetzt müssen Sie das "falsche" File aufbauen. Beispiel:

```
5ØØØ  FOR w = 316ØØ TO 323Ø3
5Ø1Ø  IF w < 319ØØ THEN POKE w, 48
5Ø2Ø  IF w >= 319ØØ THEN POKE w, 32
5Ø3Ø  NEXT w
```

Tippen Sie GO TO 5ØØØ (und warten Sie!), um das aufzubauen; dann holen Sie ein Display auf den Bildschirm (mit LIST geht das leicht) und geben Sie RANDOMIZE USR 323Ø4 ein. Sie erhalten entsprechend den durch das Programm mit POKE bei 5ØØØ eingegebenen Attributen gelbe und grüne Flächen. Sicherlich fallen Ihnen aufregendere BASIC-Routinen für die Aufgabe des falschen File ein (Streifen? Karomuster?)

Wo es anging . . .

An dieser Stelle sollten Sie erkennen können, was die Einführungsprogramme in Kapitel 1 geleistet – und wie sie das getan haben. Die Maschinencodebefehle in diesen Programmen sind in den DATA-Anweisungen enthalten und waren wegen der leichteren Eingabe in Dezimal, nicht in Hex, geschrieben. Wenn Sie die Dezimalzahlen in Hexzahlen übertragen und sie nachschlagen (die Opcode-Tabelle im Handbuch, Seite 183, führt sie numerisch geordnet auf), sehen Sie, daß sie alle Blockübertragungen in das Attribut- oder in das Displayfile betreffen. Die übertragenen Bytes werden aus dem ROM genommen, sehen meistens also recht willkürlich aus. Einige Teile des ROM (welche?) sind aber stärker strukturiert, und daher kommen die gemusterten Displays.

Seitwärtsrollen von Attributen

Versetzen Sie im LADER-Programm Zeile 1Ø wieder in den vorherigen Zustand (31999); so viel Speicherplatz brauchen wir jetzt nicht.

Die Attribute vorauszubestimmen, ist eine sehr einfache Verwendung von LDIR. Hier eine etwas raffiniertere: eine Reihe von Attributen um eine Stelle nach links wegrollen und die links außen an die rechte Seite setzen, als wäre der Bildschirm wie ein Zylinder gewickelt. Der Einfachheit halber nehme ich Reihe Ø.

```
           LD HL, A-File    21ØØ58
Schleife:  LD D, H          54
           LD E, L          5D
           INC HL           23
           LD BC, 31 dez    Ø11FØØ
           LD A, (DE)       1A
           LDIR             EDBØ
           LD (DE), A       12
```

Legen Sie etwas fest, um damit zu testen:

```
6ØØØ  FOR s = Ø TO 31: PAPER s/6: PRINT AT Ø, s; "□";: NEXT s
```

und dann RANDOMIZE USR 32ØØØ – verfolgen Sie das Abrollen. Damit sich richtig etwas tut, bauen Sie eine BASIC-Schleife

FOR t = 1 TO 5ØØ: RANDOMIZE USR 32ØØØ: NEXT t

und können zusehen, wie das herumsaust!

Wenn Sie bei geeigneten Startbedingungen das 22mal durch die Schleife laufen lassen, können Sie einen ganzen Bildschirm voller Attribute seitwärts rollen. Ich nehme ein ganz ähnliches Problem: eine Zeile des *Display*file seitwärts rollen. Dazu gehört schon eine Schleife.

Display, seitwärts gerollt

Hier der Code:

	LD A, 8 dez	3EØ8	Schleifenzähler
	LD DE, Anfang	11ØØ4Ø	Zeilenanfang
setzen:	LD H, D	62	
	LD L, E	6B	
	INC HL	23	
	PUSH DE	D5	Zeilenanfang sichern
	LD BC, 31 dez	Ø11FØØ	
	PUSH AF	F5	Zähler sichern
	LD A, (DE)	1A	erstes Byte sichern
	LDIR	EDBØ	links abrollen
	LD (DE), A	12	erstes Byte neu laden
	POP AF	F1	Schleifenzählung wiederherstellen
	DEC A	3D	Fortgangszählung
	CP B	B8	auf Ende prüfen
	JRZ Übersprung	28Ø4	
	POP DE	D1	
	INC D	14	nächste Zeile
	JR setzen	18EB	
Übersprung:	POP DE	D1	

Betten Sie das in eine weitere Schleife (oder Schleifen, Block für Block ebenso wie in einem 8 Reihen-Block) ein, dann haben Sie eine Routine, mit der das ganze Display seitlich weggerollt werden kann. Offensichtlich sind viele Abwandlungen möglich. Sehen Sie sich das genau an und stellen Sie fest, was für Veränderungen man probieren könnte.

18 Ein paar Dinge, von denen ich noch nicht gesprochen habe

Damit sind die meisten der Z8Ø-Befehle abgehandelt, soweit sie nicht Peripheriegeräte betreffen. Aber für einige Punkte ist schon noch Platz . . .

Vorweg – der Spruch weiter vorne stammt von Humty-Dumpty.

Bit-Kippen

Es gibt ein paar einfache Befehle zur Veränderung von Bits, die Sie wohl als nützlich empfinden werden. Der erste ist

```
CCF         3F
```

der die Übertragsflagge von Ø zu 1 oder umgekehrt verändert. Wenn Sie die Übertragsflagge auf 1 setzen wollen, nehmen Sie

```
SCF         37
```

Zum *Zurücksetzen* der Übertragsflagge (auf Ø) werden sie abwechselnd verwendet, zuerst SCF, dann CCF.

Zum Komplementieren des gesamten A-Registers (also jedes Bit zum umgekehrten Wert kippen) verwenden Sie

```
CPL         2F
```

Mnemonik

Als nächstes sollte ich erwähnen, daß manche Leute etwas andere Mnemonik-Opcodes verwenden als die von mir beschriebenen. Wo ich etwa LD A, (nn) schreibe, verwenden andere Leute LD (nn). Das liegt daran, daß das A-Register das einzige Register ist, das direkt geladen werden kann, so daß es streng genommen nicht angegeben werden muß. Ich halte es aber für eine nützliche Gedächtnissütze, es jedesmal zu nennen.

Alternativregister

Ich sprach davon, daß es einen zweiten Satz von Registern gibt und habe mich dann prompt nicht mehr damit befaßt. Sie können jederzeit auch ohne sie auskommen, und sie sind ohnehin nicht besonders nützlich. Rechnen kann man

mit ihnen nicht. Ihre Hauptaufgabe besteht darin, vorübergehend den Inhalt der Hauptregister zu sichern, während Sie irgendeine Routine ausführen, die den Hauptregisterinhalt auf eine Weise verändert, die Sie nicht brauchen können. Das geschieht durch Austausch des Inhalts von Haupt- und Zweitregistern vor der störenden Routine und erneut danach:

```
EX AF, AF'  Ø8      AF mit AF' tauschen
EXX         D9      BC, DE, HL mit BC', DE', HL' tauschen
CALL...     CD--    störende Routine aufrufen
EX AF, AF'  Ø8   ]
                 ]- Register wiederherstellen
EXX         D9   ]
```

Dasselbe würden Sie natürlich bewirken, wenn Sie den Registerinhalt, den Sie sichern wollen, vor dem CALL mit PUSH auf den Stapel schieben und nachher mit POP wieder abnehmen.

Nie das IY-Register verwenden. Der Spectrum braucht es!

ROM-Routinen

Ich muß auch zugeben, daß ich hin und wieder das Rad neu erfunden habe. Es ist nämlich so, daß der BASIC-Interpreter im ROM solche Routinen, wie wir sie entwickelt haben, benutzen muß. Warum sie dann nicht einfach *aufrufen*, statt sie selbst zu schreiben? Die Antwort ist im Prinzip die, daß das viel vernünftiger gewesen wäre, weil es viel Mühe, und, was fast genauso wichtig ist, Speicherplatz spart. *Aber* – was dieses Buch angeht, bestand mein Ziel darin, Ihnen Z8Ø-Maschinencode nahezubringen und, soweit möglich, die besonderen Eigenheiten des Spectrum zu meiden. Hätten alle Beispiele aus einer Reihe Aufrufen von Adressen im ROM bestanden, wäre für Sie nicht viel zu lernen gewesen!

BASIC und Maschinencode sicherstellen

Angenommen, Sie haben ein BASIC-Programm, das eine über RAMTOP (sagen wir, bei 32ØØØ) gespeicherte Maschinencode-Routine nutzt. Wie können Sie beide sinnvoll durch SAVE sichern?

Ein Weg ist der, nacheinander zu nehmen:

```
SAVE "Programm"

SAVE "M-Code" CODE 32ØØØ, 6ØØ
```

(dabei ist 6ØØ die Länge des Maschinencode-Bereichs; die genaue Länge des Codes anzugeben, ist effizienter – aber das ist ihre Sache). Dann müssen Sie wieder laden mit

```
LOAD "Programm"
```

und, wenn das im Gerät ist,

LOAD "M-Code" CODE

Vergessen Sie auch nicht CLEAR 32ØØØ.

Noch besser ist, eine kleine BASIC-Routine in das Programm zu schreiben, die das für Sie alles erledigt. Suchen Sie eine freie Zeile – etwa am Ende – und schreiben Sie

9800 LOAD "m-Code" CODE

oder was die Anfangszeile des BASIC-Programms ist. Dann geben Sie direkt ein

SAVE "Programm" LINE 98ØØ

SAVE "M-Code" CODE 32ØØØ, 6ØØ

Sobald Sie, wie gewohnt, tippen

LOAD "Programm"

wird geladen, ab Zeile 98ØØ gefahren, der Maschinencode geladen und weitergemacht.

Sie können sogar auch das SAVE als BASIC-Programm schreiben, wenn Sie wollen. Experimentieren Sie.

Vielfachroutinen

Sie können mehrere verschiedene Maschinencode-Routinen aneinanderhängen, vorausgesetzt, Sie setzen an jedes Ende einen RET-Befehl, und jede kann mit RONDOMIZE USR (Startadresse der gewünschten Routine) aufgerufen werden. Beachten Sie, daß Sie den ganzen Block derartigen Maschinencodes auf einmal sichern können; es ist *nicht* notwendig, jede Routine einzeln sicherzustellen.

Wirksame Verwendung von Maschinencode

Ich möchte zwei zu Beginn kurz angesprochene Punkte zu Ende führen. Ich sagte, es könnte andere Gründe dafür geben, daß ein Maschinencode-Programm schneller läuft als BASIC, außer daß BASIC bei jeder Ausführung von Befehlen diese interpretieren muß. Ich will das an einem Beispiel erklären:

BASIC		*Maschinencode*
1Ø FOR i = 2Ø TO 1 STEP − 1		LD B, 14
....	Schleife:	....
5Ø NEXT i		DJNZ Schleife

In beiden Fällen wird bei jeder Schleifenausführung eine Variable um 1 dekrementiert. Dieser Prozeß ist in BASIC aber viel komplizierter als in Maschinencode. Der Grund: Da BASIC teilweise mit Dezimalwerten zu tun hat, geht es davon aus, daß es das ununterbrochen tut, und zieht also 1.ØØØØØØØØ ab, was nicht einfacher ist, als 1.58712684 abzuziehen. Das verwendete Verfahren ist sogar recht kompliziert und zeitraubend. Der Maschinencode dagegen verwendet einen einzelnen, zweckbestimmten Befehl. Das geht um die 1ØØmal schneller.

Der andere Punkt, den ich aufgeschoben hatte, war, daß Maschinencode mehr Speicherplatz besetzen kann als die Entsprechung in BASIC. Hier ein Beispiel zur Erklärung:

BASIC	*Maschinencode*	*Zahl der Bytes*
3Ø IF 3 = p AND p = q THEN		
LET p = w	LD HL, 5ØØØ	3
	LD A, (HL)	1
	LD HL, 5ØØ1	3
	SUB A, (HL)	1
	JRNZ nächstbit	2
	LD HL, 5ØØ1	3
	LD A, (HL)	1
	LD HL, 5ØØ	3
	SUB A, (HL)	1
	JRNZ nächstbit	2
	LD HL, 5ØØ1	3
	LD A, (5ØØ3)	3
	LD (HL), A	1
		27 gesamt

Der Maschinencode geht davon aus, daß r, p, q und w in den Bytes 5ØØØ, 5ØØ1, 5ØØ2 und 5ØØ3 enthalten sind. In der Praxis wäre das nicht so einfach, weil jede Zahl 5 Bytes besetzt und das SUB in Wirklichkeit ein CALL zu einer Fließpunkt-Subtraktionsroutine ist. Jedenfalls würde der eigentliche Code mindestens diese 27 nachgewiesenen Bytes und vermutlich sogar mehr brauchen. Die entsprechende BASIC-Zeile benötigt nur 18 Bytes, für jedes Symbol (IF, =, w, AND und so weiter) 1, 4 für die Zeilennummer und 1 für den Zeilenbegrenzer. Je komplexer der BASIC-Befehl, desto mehr Speicherbedarf bei der Maschinencode-Version.

Wo Maschinencode noch gespeichert werden kann

Der Hauptnachteil bei der Speicherung von Code über RAMTOP ist der, daß man ihn nicht direkt als Teil eines BASIC-Programms sichern kann. Der Vorteil ist, daß man darunter anderes laden *kann*. Aber es gibt Alternativen.

Ein beliebter Trick ist der, das Ganze in einer REM-Anweisung zu speichern, der ersten Zeile des BASIC-Programms. Das erste Zeichen nach dem REM hat normalerweise die Adresse 23755. Sie beginnen Ihr BASIC-Programm also mit

```
1 REM XXXXXX . . .X
```

mit so vielen X, daß der Code hineinpaßt, dann geben Sie mit POKE den Maschinencode ein. Der Code wird zugänglich durch RANDOMIZE USR 23755 (oder LET y = USR 23755 etc.) und kann gesichert werden; außerdem wird er durch RUN nicht gelöscht.

Ein anderer Platz, wo man den Code speichern kann, ist ein Zeichenstring, der (über die Systemvariable VARS) leicht zu finden ist, vorausgesetzt, Sie machen ihn zur ersten vereinbarten Variablen – fangen also mit einem Stringarray an, der groß genug ist

```
1 DIM a$ (79)            soll 79 Bytes enthalten
```

Und setzen dann den Maschinencode in a$ (1), a$ (2) usw., während Sie den String aufbauen. Der Hauptnachteil dabei: RUN oder CLEAR löscht Ihren Code. Und die Startadresse kann sich manchmal verirren. Sie können auch als DATA speichern und wie in Kapitel 1 als Teil des BASIC-Programms über RAMTOP laden, was aber viel Platz vergeudet.

Debugging

Im Maschinencode gibt es keine eingebauten Debugging-Einrichtungen. HELPA unterstützt Sie zwar beim Redigieren von Code, ist aber eigentlich nicht für Debugging eingerichtet. (Wie mehrere andere Programme, für die mit der Behauptung geworben wird, sie seien nützlich beim "Debugging" von Maschinencode!) In diesem Stadium ist das Beste für Sie, die Routine mit Bleistift und Papier als *Schreibtischtest* auszuprobieren (siehe ZX Spectrum – Programmieren leicht gemacht, S. 53). Natürlich können Sie Verfolgungsanweisungen in den Maschinencode einbauen, aber achten Sie auf Veränderungen in Adressen und Sprunggrößen.

Ein nützlicher (obschon auf den ersten Blick plumper) Trick ist der, die Routine zuerst in BASIC zu schreiben und *diese* von Fehlern zu befreien. Verwenden Sie dabei nur BASIC-Befehle, die dem Maschinencode entsprechen. (Das heißt, Sie müssen den Maschinencode in BASIC *nachahmen*.) Das geht, wenn überhaupt, langsam, aber das Ziel läßt sich erreichen.

Für die wahrhaft Ehrgeizigen legt das eine Aufgabe nahe. HELPA soll durch Anfügen einer Routine SINGLE-STEP (Einzelschritt) verbessert werden, die das Programm Befehl für Befehl durchgeht und die Register auf dem

Bildschirm anzeigt. Sie müssen a) eine Maschinencode-Routine schreiben, um alle Register mit PUSH auf den Stapel zu setzen und sie dann in Hex auf dem Bildschirm anzuzeigen, b) eine Routine anfügen, die eine Keyboard-Eingabe verlangt, c) zwischen jede Zeile Maschinencode ein CALL für diese Routine schreiben (um anzuzeigen wo, verwenden Sie die **-Begrenzer in HELPA), und d) erarbeiten, wie Sie zu BASIC zurückkehren können, wenn Sie wollen.

Anhang 1
Umwandlung Hex/Dezimal

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
8	−128	−127	−126	−125	−124	−123	−122	−121	−120	−119	−118	−117	−116	−115	−114	−113
9	−112	−111	−110	−109	−108	−107	−106	−105	−104	−103	−102	−101	−100	−99	−98	−97
A	−96	−95	−94	−93	−92	−91	−90	−89	−88	−87	−86	−85	−84	−83	−82	−81
B	−80	−79	−78	−77	−76	−75	−74	−73	−72	−71	−70	−69	−68	−67	−66	−65
C	−64	−63	−62	−61	−60	−59	−58	−57	−56	−55	−54	−53	−52	−51	−50	−49
D	−48	−47	−46	−45	−44	−43	−42	−41	−40	−39	−38	−37	−36	−35	−34	−33
E	−32	−31	−30	−29	−28	−27	−26	−25	−24	−23	−22	−21	−20	−19	−18	−17
F	−16	−15	−14	−13	−12	−11	−10	−9	−8	−7	−6	−5	−4	−3	−2	−1
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
E	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Zweierkomplement

gewöhnlich

Anhang 2
Speicherreservierungs-Tabellen

(siehe Beilagekarte)

Anhang 3
Adressen von Systemvariablen

(siehe Beilagekarte)

Anhang 4
Zusammenfassung von Z80-Befehlen

Das ist eine Liste der gesamten Opcode-Mnemonik mit einer Zusammenfassung der Wirkungen. Befehle, die im Text genauer erläutert werden, sind mit einem Seitenhinweis versehen. Die Auswirkungen auf die Flaggen habe ich weggelassen; dazu siehe die Zilog Reference Card oder die in der Bibliographie unter "Maschinencode" genannten Bücher.

ADC	Seite 58	Addiere einschließlich Übertragsflagge. Speichere in A oder HL.
ADD	Seite 49	Addiere und laß Übertragsflagge unbeachtet. Speichere in A oder HL.
AND	Seite 54	Logisches AND auf zugehörigen Bits: Speichere in A.
BIT	Seite 65	BIT b, r setzt die Nullflagge entsprechend dem Wert des b-ten Bits des Bytes in Register r. Die Bits haben in jedem Byte die Reihenfolge 7654321Ø.
Call	Seite 29	Ruft eine Subroutine auf. Bedingte Aufrufe werden signalisiert durch die zusätzlichen Buchstaben C (rufe auf, wenn die Übertragsflagge gesetzt ist); M (wenn die Vorzeichenflagge gesetzt ist – "das Ergebnis [eines Vergleichs] ist negativ"); NC (wenn die Übertragsflagge nicht gesetzt ist); NZ (wenn die Nullflagge nicht gesetzt ist); P (wenn die Vorzeichenflagge nicht gesetzt ist – "das Ergebnis ist positiv"); PE (wenn die Paritätsflagge gesetzt ist; dies unbeachtet lassen); PO (wenn die Paritätsflagge nicht gesetzt ist; ebenfalls unbeachtet lassen); Z (wenn die Nullflagge gesetzt ist). Wegen Flaggen siehe Seiten 92, 114.
CCF	Seite 103	Komplement-Übertragsflagge (d. i. Ø und 1 austauschen).
CP	Seite 55	Vergleiche: Setzt die Flaggen wie eine Subtraktion von A, läßt A aber unverändert.
CPD		Vergleiche und dekrementiere. Vergleiche durch HL, dann dekrementiere HL und BC.
CPDR	Seite 99	Vergleiche, dekrementiere, wiederhole: Blocksuche. Wie CPD, wiederholt aber, bis Resultat des Vergleichs entweder Ø oder bis BC Ø erreicht.
CPI		Wie CPD, nur inkrementiert HL; BC dekrementiert weiterhin.
CPIR	Seite 99	Wie CPDR, inkrementiert aber HL.
CPL	Seite 103	Komplement (Kippbits) des A-Registers.
DAA		Dezimalanpassungs-Akkumulator. Verwendet bei Arbeit mit binär codierten Dezimalzahlen; nicht beachten.
DEC	Seite 56	Dekrementiere: verringert Wert um 1.
DI		Unterbrechungen stillegen. Nicht beachten.

DJNZ	Seite 58	Dekrementiere, spring, wenn nicht Null. Dekrementiert B und springt relativ, es sei denn, Nullflagge ist gesetzt. Verwendet in Schleifen wie FOR/NEXT in BASIC.
EI		Ermögliche Unterbrechungen. Nicht beachten.
EX	Seite 104	Tausche Werte aus. Befehle mit (SP) tauschen die Register HL, IX oder IY mit der Oberseite des Stapels.
EXX	Seite 104	Tausche alle drei Registerpaare BC, DE, HL mit ihren Ergänzungen BC', DE', HL'.
HALT		Warte auf Unterbrechung. Falls Sie nicht Hardware angeschlossen haben und wirklich wissen, was Sie tun, NICHT VERWENDEN, weil das Programm ewig wartet.
IM		Interrupt mode = Unterbrechungsmodus. Nicht beachten.
IN		Eingabe von einem Gerät. Nicht beachten.
INC	Seite 56	Inkrementiere: erhöht Wert um 1.
IND, INDR, INI, INIR		Eingabebefehle entsprechend LDD, LDDR, LDI, LDIR. Nicht beachten.
JP	Seite 55	Springe. Varianten mit zusätzlich C, M, NC, NZ, P, PE, PO, Z sind bedingte Sprünge mit Bedingungen wie bei CALL.
JR	Seite 55	Spring relativ – gefolgt von einer 1 Byte-Verschiebung. Bedingungsvarianten sind nur C, NC, NZ, Z.
LD	Seite 39	Lade. Kann alle fünf Adressierarten verwenden.
LDD		Nicht dasselbe wie LD D! Lade, worauf HL zeigt, in das, worauf DE zeigt, dekrementiere BC, DE, HL.
LDDR	Seite 100	Lade, dekrementiere, wiederhole: Blockübertragung. Leiste LDD, bis BC Null erreicht. Kopiert einen Speicherblock, dessen Länge in BC gespeichert ist, aus dem, worauf HL zeigt, in das, worauf DE zeigt.
LDI		Wie LDD, nur inkrementieren HL und DE. BC dekrementiert weiter.
LDIR	Seite 100	Wie LDDR, nur inkrementieren HL und DE.
NEG		Negativ: verändere das Vorzeichen des Inhalts von A.
NOP		Keine Operation. Tu einen Zeitzyklus lang nichts – das heißt, vergeude Zeit. Nützlich für zeitweiliges Löschen von Befehlen beim Debugging; harmlos und hilfreich.
OR	Seite 54	Logisches OR auf Bits. In A speichern.
OTDR, OTIR, OUT, OUTD, OUTI		Verschiedene Ausgaben. Nicht beachten.
POP	Seite 59	Von Stapel auf angegebenes Register setzen.
PUSH	Seite 59	Von Register auf Stapel setzen.
RES	Seite 76	Ein Bit zurücksetzen – also zu Null machen.
RET	Seite 31	Von Subroutine zurückspringen. Bedingte Rücksprünge, den Möglichkeiten für CALL entsprechend, sind möglich. (Bedingungen bei einem CALL müssen denen bei einem RET nicht entsprechen!)
RETI, RETN		Spring von Unterbrechungs-Routinen zurück. Nicht beachten.
RL		Nach links rotieren: wie eine Verschiebung, nur ist Flagge eingeschlossen, als wäre sie Bit 8.

RLA		Akkumulator nach links rotieren. Wie RL A, aber mit anderer Wirkung auf die Flaggen.
RLC		Nicht dasselbe wie RL C! Rotiere nach links, aber setz Bit 7 in Übertrag *und* in Bit Ø.
RLCA		Wie RLC A, aber derselbe Flaggenunterschied wie bei RLA.
RLD		Ganz und gar nicht das Erwartete: Rotiere nach links dezimal. Verwendet für binär codierte Dezimalzahlen. Nicht beachten.
RR		Wie RL, aber nach rechts.
RRA		Wie RLA.
RRC		Wie RLC.
RRCA		Wie RLCA.
RRD		Wie RLD.
RST		Wie CALL, aber nur von den Adressen Ø, 8, 1Ø, 18, 2Ø, 28, 3Ø, 38 (hex) aus. Diese sind im Spectrum alle im ROM; siehe Ian Logans Bücher in der Bibliographie. RST Ø wirkt so, als schalte man zeitweilig den Strom ab.
SBC	Seite 58	Subtrahiere und berücksichtige Übertragsflagge. Speichere in A oder HL.
SCF	Seite 103	Setze Übertragsflagge (auf 1).
SET	Seite 76	Setze ein Bit – das heißt, mach es zu 1.
SLA	Seite 58	Verschiebe links arithmetisch. Alle Bits um 1 erhöht; Bit Ø wird Ø.
SRA	Seite 58	Verschiebe rechts arithmetisch. Nimm Bits um 1 herunter; kopiere Bit 7 in 6 *und* 7.
SRL	Seite 58	Logische Rechtsverschiebung. Alle Bits einen Platz nach rechts schieben. Bit 7 wird Ø.
SUB	Seite 49	Subtrahiere und beachte Übertrag nicht. Speichere in A. (Es gibt keinen Befehl SUB HL, r. Wenn Sie einen wollen, setzen Sie die Übertragsflagge zurück und verwenden Sie SBC.)
XOR	Seite 54	Exklusives OR auf jedem Bit. Speichere in A.

Anhang 5
Null- und Übertrags-Flaggen

(siehe Beilagekarte)

Anhang 6
Z80-Opcodes

Das ist eine komplette Liste von Z80-Opcodes, nach der Mnemonik alphabetisch geordnet. In der Liste steht das Symbol n für jede Einzelbyte-Zahl; nn für jede 2 Byte-Zahl; und d ist eine Verschiebung um 1 Byte, geschrieben in Zweierkomplement-Notation. Beachten Sie, daß alle 2 Byte-Zahlen so codiert sind, daß das jüngere Byte *vor* dem älteren kommt.

Beispiele:

LD BC, nn hat Opcode Ø1 nn, so daß LD BC, 732F codiert wird als Ø12F73.

LD A, (IY + d) hat Opcode FD7E, so daß LD A (IY + Ø7) codiert wird als FD7EØ7.

Die Tabelle der Opcodes beruht auf einer von Zilog Inc. veröffentlichten. Ein numerisches Listing nach Opcodes enthält Anhang A des Sinclair-Handbuchs. Beachten Sie dort die Verwendung von Kleinbuchstaben für die Mnemonik.

Liste siehe Beilagekarte

Anhang 7
HELPA

Das ist ein vielseitiges Nutzprogramm in BASIC für leichte Abänderung, damit Sie Maschinencode redigieren, laden und fahren können. Die Abkürzung bedeutet Hex Editor, Loader und Partial Assembler (Hex-Editor, Lader und Teilassembler). Das Programm beruht auf einem ZX81-Programm in "Maschinencode und besseres BASIC", ist aber verändert worden, um die verbesserten Eigenschaften des Spectrum zu berücksichtigen.

1. Geben Sie es ein und sichern Sie mit

SAVE "helpa" LINE 5

sobald Sie sicher sind, daß Sie keine Fehler gemacht haben.

2. Fahren Sie mit RUN. Es wird die Menge an Speicherplatz anfordern, die reserviert werden soll. Zuerst „Wollen Sie nichtüblichen Speicher?" mit einer Eingabe beantworten. Alles außer ENTER wird ausgelegt als Frage nach einem anderen Startwert für den Maschinencode als den im Buch durchgehend verwendeten üblichen 32ØØØ. Dann wird der Speicher mit CLEAR gelöscht, RAMTOP zurückgesetzt und zur Prüfung der neue RAMTOP-Wert ebenso wie die Startadresse für den Maschinencode angezeigt.

3. Dann verlangt es die Zahl der Datenbytes, die dem Programmbereich vorangehen. Sie müssen mit POKE später eingegeben werden (vielleicht wollen Sie dafür eine eigene Routine anfügen).

4. Danach zeigt es diesen Wert und die tatsächliche Startadresse an, die für den Maschinencode verwendet wird.

5. Anschließend verlangt es den HEXCODE und liefert einen roten Cursor, ein blinkendes ⊳-Zeichen.

6. Sie können jetzt Maschinencode in Hex eingeben (und mit den unten beschriebenen Steuerbefehlen manipulieren). Bei Eingabe jeder Codegruppe nimmt das Programm alle Leerräume weg (Sie können diese also einsetzen, wie Sie wollen), teilt in Zweizeichen-Codes auf und zeigt diese in 1Ø Kolonnen an. Am Ende jeder Gruppe wird ein abschließender **-Begrenzer angefügt, der Cursor geht zum Ende des eben eingegebenen Codes.

Beispielsweise wird die Eingabe "□a1□e23□□4" angezeigt als

a1 e2 34 ** >

(genau wie „a1e234" oder "a1□e2□34").

7. Nun kann die nächste Codegruppe eingegeben werden. Sie wird automatisch unmittelbar an die Cursorposition angehängt. Die ** dienen nur zur Bequemlichkeit des Anwenders und bleiben unbeachtet, wenn der Code über RAMTOP geladen wird, so daß die Gruppen Z80-Befehlsgruppen *nicht* entsprechen müssen; das hilft aber beim Überprüfen.

8. Zusätzlich können "Steuer"-Befehle eingegeben werden. Diese *müssen* das erste Symbol in ihrer Gruppe sein; nachfolgende Zeichen werden in der Regel nicht beachtet, ausgenommen einige unten beschriebene Fälle.

Die Steuerbefehle sind:

g	(Go)	Fahr die Maschinencode-Routine
l	(Load)	Lade die Maschinencode-Routine über RAMTOP
m	(Move)	Bewege Cursor vorwärts
n	(Negative)	Bewege Cursor rückwärts
p	(Print)	Zeige laufendes Hexlisting an
r	(Relative)	Verwendet für automatische Berechnung relativer Sprünge
s	(Save)	Sichere Programm
x	(eXcise)	Lösche aus dem Listing

Genauere Beschreibung anschließend in einer praktischeren Reihenfolge:

m, n Der Befehl ma, wobei a eine Zahl ist, bewegt den Cursor a Leerräume vorwärts. na bewegt ihn um a Leerräume zurück. Er ist dagegen geschützt, über das Listing hinauszurutschen. Fehlt a, wird es auf 1 gesetzt.

x Ein Befehl xa, bei dem a eine Zahl ≥ Ø ist, löscht die nächsten a Zeichenpaare (einschließlich Doppel-**) nach dem Cursor. Das Display bleibt unberührt, bis p gedrückt wird. Fehlt a, wird es auf 1 gesetzt.

p Zeigt den laufenden Text an und bewegt den Cursor an den Beginn.

s Sichert das Programm einschließlich des laufenden Hexlistings. Eine Option erlaubt Ihnen, das Programm zu benennen, wie Sie wollen. Damit Sie ein gesichertes Programm fahren können, sichern Sie mit SAVE, *bevor* "l" und „g" gedrückt werden. Dann mit LOAD nach BASIC laden und GO TO 2ØØ eingeben (nicht RUN!), dann "l", dann „g".

l Lädt den Maschinencode, befreit von überflüssigen **-Begrenzern, über RAMTOP, beginnend bei dem von Ihnen gesetzten RAMTOP-Wert; fügt den abschließenden RET-Befehl an.

g Das fährt die Routine. Steuerung kehrt zu HELPA zurück (falls kein Absturz eintritt!)

r Eine nützliche Einrichtung, um *relative* Sprünge zu erleichtern. Von Hand sind sie mühsam, weil man die Verschiebungen berechnen und sie in Zweierkomplement-Hex setzen muß, aber angenehm für Programme wegen der Vereinfachung. Das geht so:

a) Geben Sie die Maschinencode-Routine ein und setzen Sie alle relativen Sprunggrößen auf ØØ.

b) Um den richtigen Wert für einen Sprung zu verändern, setzen Sie den Cursor unmittelbar vor die ØØ und löschen durch Druck auf "x". Geben Sie nun rn ein, wobei die Zahl n die (dezimale) Position des *Ziel*bytes für den Sprung ist, vom Bildschirmdisplay wie folgt abgelesen. Numerieren Sie die Reihen und Spalten des Hexcode-Arrays ab Ø so:

```
  Ø 1 2 3 4 4 5 6 7 8 9
Ø
1
2
3
.
.
.
```

und setzen Sie n = xy für Reihe x, Spalte y. (Das heißt, für das Byte in Reihe 17, Spalte 5 geben Sie r175 ein.) Die Tatsache, daß es 1Ø Spalten gibt, erleichtert das Ablesen der Zahlen. Sie könnten das Programm so verändern, daß der Cursor zum Ziel bewegt und der Sprung von dort aus gefunden wird. In der Praxis geht das langsamer, weil der Cursor dauernd verschoben werden muß.

c) Drücken Sie nun "p". Sie erhalten ein revidiertes Display einschließlich der richtigen Sprunggröße.

d) Stellen Sie den Cursor vor die nächste relative Sprunggröße und wiederholen Sie.

e) Beachten Sie, daß das Programm automatisch alle vorhandenen ** berücksichtigt und den erforderlichen Zweierkomplement-Code liefert. Falls der Sprung über den zulässigen Bereich hinausgeht, teilt es das mit.

f) Das klingt vielleicht kompliziert, deshalb hier ein Beispiel unter Verwendung der Spalten-scrolling-Routine aus Kapitel 14. Geben Sie der Reihe nach die Hexcode-Gruppen ein. Das Ergebnis wird so aussehen:

```
Ramtop:         31999
MC-Bereich:     32ØØØ
Daten:          32ØØØ bis 31999
Run USR:        32ØØØ
                HEXCODE:
                Ø12ØØØ**dd211f58**3e
                15**dd562Ø**dd72ØØ**
                ddØ9**3d**b8**2ØØØ**
                >            ‾‾
```

ØØ unterstrichen ist die Größe des zu berechnenden relativen Sprungs. (Der Datenbereich sieht ein bißchen merkwürdig aus – Sie könnten das Programm verändern, daß es anzeigt "Daten: keine", wenn die Zahl der Datenbytes Ø beträgt.)

Mit „n" setzen Sie den Cursor so vor die letzten ØØ:

ddØ9**3d**b8**2Ø [>] ØØ**

und tippen dann "x", um die ØØ zu löschen. Der Befehl hier lautet JRNZ *Schleife*, und die Schleife beginnt beim Code "dd" in der zweiten Hexzeile. Das ist in Reihe 1, Spalte 2 (nicht vergessen, beide beginnen mit Ø), so daß Sie eingeben müssen "r12". Tun Sie das.

```
Ramtop:    31999
m/c area:32000
Data:      32000 to 31999
Run USR:  32000
HEX CODE:
 01 20 00 ** dd 21 1f 58 ** 3e
 15 ** dd 56 20 ** dd 72 00 **
 dd 09 ** 3d ** b8 ** 20 00 **
```

Abbildung A7.1
HELPA vor einem relativen Sprung.

```
Ramtop:    31999
m/c area:32000
Data:      32000 to 31999
Run USR:  32000
HEX CODE:
 01 20 00 ** dd 21 1f 58 ** 3e
 15 ** dd 56 20 ** dd 72 00 **
 dd 09 ** 3d ** b8 ** 20 f4 **
>
```

Abbildung A7.2
Die Sprungadresse f4 ist eingefügt worden.

Kurz danach wird der Bildschirm leer, dann zeigt er ein neues Listing an. ØØ ist ersetzt durch f4, die richtige relative Sprunggröße.

Hier das vollständige Listing:

```
  5  REM helpa © 1982 Ian Stewart & Robin Jones
  7  POKE 236Ø9, 5Ø
 1Ø  INPUT "Wollen Sie nichtueblichen Speicher?"; a$
 2Ø  IF a$ <> "" THEN INPUT "START MC-Bereich?"; rt
 30  IF a$ = "" THEN LET rt = 32ØØØ
 4Ø  CLEAR rt − 1
 5Ø  LET rt = PEEK 2373Ø + 256 * PEEK 23731 + 1
 6Ø  PRINT "Ramtop: □ □"; rt − 1
 7Ø  PRINT "MC-Bereich: □"; rt
 8Ø  INPUT "Zahl der Datenbytes?"; d
 9Ø  PRINT "Daten: □ □ □ □"; rt; "□ bis □"; rt + d − 1
1ØØ  PRINT "Run USR: □"; rt + d
11Ø  PRINT PAPER 6; "HEX CODE:"
12Ø  LET ci = Ø
13Ø  LET h$ = ""
14Ø  GO SUB 4ØØ
15Ø  DIM f(2Ø)
151  LET f(1) = 7ØØ
152  LET f(6) = 8ØØ
153  LET f(7) = 9ØØ
154  LET f(8) = 1ØØØ
155  LET f(10) = 11ØØ
156  LET f(11) = 12ØØ
157  LET f(12) = 13ØØ
158  LET f(13) = 14ØØ
159  LET f(14) = 15ØØ
16Ø  LET f(18) = 16ØØ
2ØØ  INPUT i$
21Ø  LET a = Ø
22Ø  LET a = a + 1
```

```
230  IF a > LEN i$ THEN GO TO 300
240  IF i$(a) <> "□" THEN GO TO 220
250  LET i$ = i$(TO a - 1) + i$(a + 1 TO)
260  GO TO 230
300  IF CODE i$(1) > = 103 THEN GO TO 500
310  LET i$ = i$ + "**"
320  LET h$ = h$(TO 2 * ci) + i$ + h$(2 * ci + 1 TO)
330  GO SUB 450
340  GO SUB 600
350  GO SUB 400
360  GO TO 200
400  REM Cursor anzeigen
410  PRINT AT 5 + INT (ci/10), 3 * (ci - 10 * INT (ci/10)); FLASH 1;
     INK 2; ">";
420  RETURN
450  REM Cursor loeschen
460  PRINT AT 5 + INT (ci/10), 3 * (ci - 10 * INT (ci/10)); "□";
470  RETURN
500  REM Keyboard-Routinen
510  GO SUB f(CODE i$(1) - 102)
520  GO TO 200
600  REM angehaengt Text anzeigen
610  FOR j = 1 TO LEN i$/2
620  PRINT i$(2 * j - 1 TO 2 * j) + "□"
630  LET ci = ci + 1
640  IF ci = 10 * INT (ci/10) THEN PRINT "□ □";
650  NEXT j
660  RETURN
700  REM los
710  CLS
720  LET y = USR(rt + d)
```

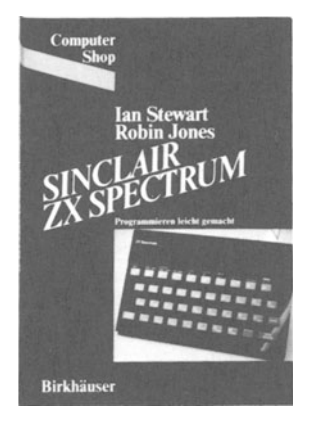
```
 730  RETURN
 800  REM ueber RAMTOP laden
 810  LET h$ = h$ + "c9"
 820  LET j = rt + d - 1
 830  LET i = - 1
 840  LET j = j + 1
 850  LET i = i + 2
 860  IF i > LEN h$ THEN RETURN
 870  IF h$(i) = "*" THEN GO TO 850
 880  POKE j, 16 * (CODE h$(i) - 48 - 39 * (h$(i) > "9")) + CODE
      h$(i + 1) - 48 - 39 * (h$(i + 1) > "9")
 890  GO TO 840
 900  REM Cursor positiv bewegen
 910  GO SUB 450
 920  IF LEN i$ = 1 THEN LET cm = 1
 930  IF LEN i$ > 1 THEN LET cm = VAL i$(2 TO)
 940  LET ci = ci + cm
 950  IF ci > LEN h$/2 THEN LET ci = LEN h$/2
 960  GO SUB 400
 970  RETURN
1000  REM Cursor negativ bewegen
1010  GO SUB 450
1020  IF LEN i$ = 1 THEN LET cm = 1
1030  IF LEN i$ > 1 THEN LET cm = VAL i$(2 TO)
1040  LET ci = ci - cm
1050  IF ci < 0 THEN LET ci = 0
1060  GO SUB 400
1070  RETURN
1100  REM anzeigen
1110  PRINT AT 5, 0
1120  FOR r = 1 TO 17: PRINT " [32 x □] ";: NEXT r
```

```
1130  PRINT AT 5, Ø; "□";
1140  LET ci = Ø
1150  FOR j = 1 TO LEN h$/2
1160  PRINT h$(2 * j − 1 TO 2 * j) + "□";
1170  LET ci = ci + 1
1180  IF ci = 1Ø * INT (ci/1Ø) THEN PRINT "□ □";
1185  NEXT j
1190  GO SUB 4ØØ
1195  RETURN
1300  REM relative Spruenge
1310  LET jci = VAL i$(2 TO)
1320  LET js = jci − ci − 1
1325  GO SUB 2ØØØ
1330  IF js > = 128 AND js < = 127 THEN GO TO 1355
1340  INPUT "Ungueltige Groesse: mit ENTER weiter"; a$
1350  RETURN
1355  IF js < Ø THEN LET js = js + 256
1360  LET xØ = INT(js/16)
1365  LET xØ = js − 16 * x1
1367  LET x$ = CHR$ (x1 + 48 + 39 * (x1 > 9)) + CHR$
      (xØ + 48 + 39 * (xØ > 9))
1370  LET h$ = h$(TO 2 * ci) + x$ + h$(2 * ci + 1 TO)
1375  LET ci = ci + 1
1380  GO SUB 11ØØ
1390  RETURN
14ØØ  REM sichern
141Ø  INPUT "SAVE Name? Vorgabe""helpa"""; n$
142Ø  IF n$ = "" THEN LET n$ = "helpa"
143Ø  POKE rt + d + LEN h$, 2Ø1
144Ø  SAVE n$ CODE rt, d + LEN h$ + 1
145Ø  PRINT "Fuer neues Laden" ' ' ",LOAD" " "; n$; "" "□ CODE"
```

```
146Ø RETURN
16ØØ REM loeschen
161Ø IF LEN i$ = 1 THEN LET k = 1
162Ø IF LEN i$ > 1 THEN LET k = VAL i$(2 TO)
163Ø LET h$ = h$(TO 2 * ci) + h$(2 * ci + 2 * k + 1 TO)
164Ø RETURN
2ØØØ REM Asteriskanpassung
2Ø1Ø IF js < Ø THEN LET w$ = h$(2 * jci + 1 TO 2 * ci)
2Ø2Ø IF js > = Ø THEN LET w$ = h$(2 * ci + 1 TO 2 * jci)
2Ø3Ø LET sc = Ø
2Ø4Ø FOR t = 1 TO LEN w$
2Ø5Ø IF w$(t) = "*" THEN LET sc = sc + 1
2Ø6Ø NEXT t
2Ø7Ø IF js < Ø THEN LET js = js + sc/2
2Ø8Ø IF js > Ø THEN LET js = js - sc/2
2Ø9Ø RETURN
```

Bibliographie

Carr, *Z80 User's Manual* Beston Publishing Co. Inc.
Logan, *Understanding Your Spectrum*, Melbourne House Group.
Nichols, Nichols, and Rony, *Z80 Microprocessor Programming and Interfacing*, Howard Sams & Co.
Parr, A Z-80 Workshop Manual, Bernard Babani LTD, London W6 7NF.
Sloan, *Introduction to Minicomputers and Microcomputers*, Addison-Wesley.
Spracklen, *Z80 and 8080 Assembly Language Programming*, Hayden.
Zaks, *Programmierung des Z80, Sybex.*
Zilog Z80 CPU Programming Reference Card and
Zilog Z80 CPU Technical Maunal, Zilog UK Ltd., Nicholson House, Maidenhead, Berks.



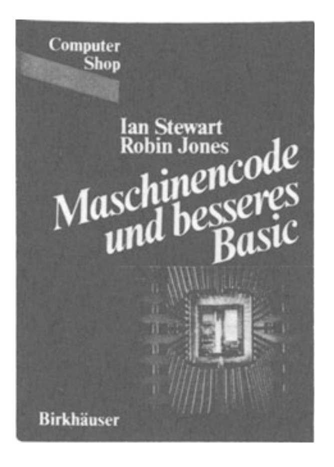
Ian Stewart
Robin Jones

Sinclair ZX Spectrum

Programmieren leicht gemacht

160 Seiten, Broschur
2. Auflage 1983

Wenn Sie sich gerade einen ZX Spectrum gekauft haben oder einen kaufen wollen, dann ist dieser Band genau das Richtige für Sie. Wir zeigen Ihnen in leicht verständlichen Schritten, wie man es anfängt, seine eigenen Programme zu schreiben. Sie finden: • Graphiken • Ketten • Daten • Methoden der Fehlersuche • Licht und Ton • Programmierstil. Die 26 Fertigprogramme – zum Beispiel für Videospiele – die am Ende des Bandes aufgeführt sind, brauchen Sie allesamt nur eingeben und mit RUN laufen lassen.



Ian Stewart
Robin Jones

Maschinencode und besseres Basic

240 Seiten, Broschur
2., verbesserte Auflage 1983

Dieser Folgeband zu «ZX81. Programme, Spiele, Graphik» behandelt folgende wichtige Gebiete: • Datenstrukturen – für bessere Verarbeitung • Strukturiertes Programmieren – für Programme, die auch funktionieren • Maschinencode – für ganz schnelle Abläufe • Verschiedene Anhänge – zur Unterstützung, wenn Sie in Maschinencode programmieren. Der grösste Teil des Bandes ist maschinenunabhängig für auf Z80 aufbauende Computer verwendbar. Alle Programme laufen jedoch unverändert beim Sinclair ZX81 mit dem 16K-RAM-Zusatzspeicher.